ARDiS: A Portable and Unified Resource Management Framework in Real Hardware Systems

MOHAMMED BAKR SIKAL, Karlsruhe Institute of Technology, Karlsruhe, Germany
JEFERSON GONZALEZ-GOMEZ, Instituto TecnolÃ³ogico de Costa Rica (TEC), Cartago, Costa Rica
ANDREAS NÖBEL, Karlsruhe Institute of Technology, Karlsruhe, Germany
HEBA KHDR, Karlsruhe Institute of Technology, Karlsruhe, Germany
JÖRG HENKEL, Karlsruhe Institute of Technology, Karlsruhe, Germany

Designing efficient RM strategies is a cornerstone of modern computing, driving innovations in performance optimization, energy efficiency, and security. While simulators have long been the go-to tools for RM research, they fail to balance accuracy and practicality: high-fidelity simulators are excruciatingly slow, and low-fidelity ones compromise on reliability. Real hardware offers unparalleled precision and accuracy but remains underutilized due to significant barriers, including fragmented implementations, lack of portability, and prohibitive development overhead. We present ARDiS, the first open-source[1] and portable framework to provide a unified, architecture-agnostic platform for running system-level resource management (RM) techniques directly on real hardware. ARDiS eliminates the need to "reinvent the wheel," enabling researchers to design, implement, and evaluate sophisticated RM strategies—including machine learning-based approaches—with minimal effort and maximum reproducibility. To demonstrate its versatility, we evaluate ARDiS on two real-world hardware platforms: a server-grade heterogeneous processor (Intel i9-12900) and a resource-constrained embedded system (NVIDIA Jetson TX2). Through extensive experimentation, we validate the ability of ARDiS to deliver accurate, scalable, and reproducible results across diverse platforms and application domains. By lowering the barriers to hardware-based RM research, ARDiS empowers the design automation community to explore new frontiers in system-level optimization and innovation.

CCS Concepts: • **Software and its engineering → Application specific development environments**; **Operating systems**; **Embedded software**; **Open source model**; • **Information systems →** *Process control systems*; • **Hardware → Power and energy**; **Electronic design automation**; **On-chip resource management**; • **Human-centered computing →** *Visualization*; • **Security and privacy →** *Systems security*; • **Computing methodologies → Machine learning**; *Artificial intelligence*

Additional Key Words and Phrases: Open-source framework, system-level resource management on real hardware, enabling machine learning-based resource management, portability to different architectures and platforms

[1] https://github.com/Chair-for-Embedded-Systems/ARDiS

This work was partially funded by the Federal Ministry of Education and Research, BMBF, as part of the MANNHEIM-CeCaS project (grant: 16ME0817).
Authors' Contact Information: Mohammed Bakr Sikal (corresponding author), Karlsruhe Institute of Technology, Karlsruhe, BW, Germany; e-mail: bakr.sikal@kit.edu; Jeferson Gonzalez-Gomez, Instituto Tecnolóogico de Costa Rica (TEC), Cartago, Costa Rica; e-mail: jeferson.gonzalez@kit.edu; Andreas Nöbel, Karlsruhe Institute of Technology, Karlsruhe, BW, Germany; e-mail: andreas.noebel@student.kit.edu; Heba Khdr, Karlsruhe Institute of Technology, Karlsruhe, BW, Germany; e-mail: heba.khdr@kit.edu; Jörg Henkel, Karlsruhe Institute of Technology, Karlsruhe, BW, Germany; e-mail: henkel@kit.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).
ACM 1084-4309/2026/03-ART80
https://doi.org/10.1145/3793861

ACM Reference Format:
Mohammed Bakr Sikal, Jeferson Gonzalez-Gomez, Andreas Nöbel, Heba Khdr, and Jörg Henkel. 2026. ARDiS: A Portable and Unified Resource Management Framework in Real Hardware Systems. *ACM Trans. Des. Autom. Electron. Syst.* 31, 4, Article 80 (March 2026), 24 pages. https://doi.org/10.1145/3793861

1 Introduction

The growing complexity of modern computing systems has made system-level **resource management (RM)** a critical area of research, with objectives ranging from improving performance and energy efficiency to addressing security concerns. RM techniques rely on tuning system-level parameters, i.e., RM "knobs", to optimize resource allocation under various constraints. These knobs include **Dynamic Voltage and Frequency Scaling (DVFS)**, application mapping, application migration, scheduling, and so on. Effective RM strategies are essential for achieving optimal system behavior in the face of diverse and dynamic workloads.

Traditionally, researchers have employed simulation frameworks to design and evaluate RM techniques [6, 8, 15, 27, 28, 37]. Simulation platforms can be broadly categorized into two types: high-fidelity and low-fidelity simulators. High-fidelity (e.g., cycle accurate) simulators [5, 34] offer detailed and accurate modeling of system behavior, but are computationally expensive and often impractical for exploring complex RM strategies. On the other hand, low-fidelity simulators [9, 29] are faster but lack the precision needed to advance the state of RM research. This tradeoff between simulation accuracy and computational cost has become a significant bottleneck to produce reliable insights for sophisticated RM techniques, particularly those involving **machine learning (ML)** algorithms. In fact, the application of machine learning techniques to resource management was adopted as early as over a decade ago [10] and has now become the norm in most contemporary system-level resource management approaches [21, 25, 30–32] and in the hardware security domain [13, 14, 33].

To address this, interval simulators [7, 12, 26] have emerged as an alternative, offering a compromise between accuracy and computational efficiency. Unlike traditional high-fidelity simulators that model every cycle or instruction and low-fidelity simulators that abstract system behaviors at a coarse level, interval simulators operate by modeling system execution in discrete time intervals. They rely on performance models that estimate system behavior across these intervals rather than tracking every instruction, significantly reducing computational overhead while maintaining a reasonable level of accuracy.

However, despite their advantages, interval simulators still suffer from two fundamental limitations. First, even with their reduced computational complexity, simulation times remain orders of magnitude longer compared to real hardware execution. This is highlighted in Figure 1, which shows the execution time of four single applications from the PARSEC [4] and SPLASH-2 [35] benchmark suites, executed alone on different platforms: server-grade Intel i9 Alder Lake, embedded NVIDIA Jetson TX2 board, and HotSniper [26] (running on a host server-grade AMD Ryzen 7x machine). This extended runtime can significantly hinder the exploration of complex RM strategies, especially targeting large workloads or requiring runtime adaptability. Second, although interval simulators offer improved accuracy over low-fidelity approaches, they still lack the precise modeling needed to capture complex system behaviors, particularly for workloads with highly dynamic resource demands or fine-grained optimization requirements.

A promising alternative is to implement and evaluate RM techniques directly on real hardware platforms. Unlike simulators, real hardware provides precise accuracy and eliminates many of the assumptions required in software-based models. Researchers have started to take advantage of real hardware to explore advanced RM strategies [8, 16, 22, 24], including ML-based approaches.

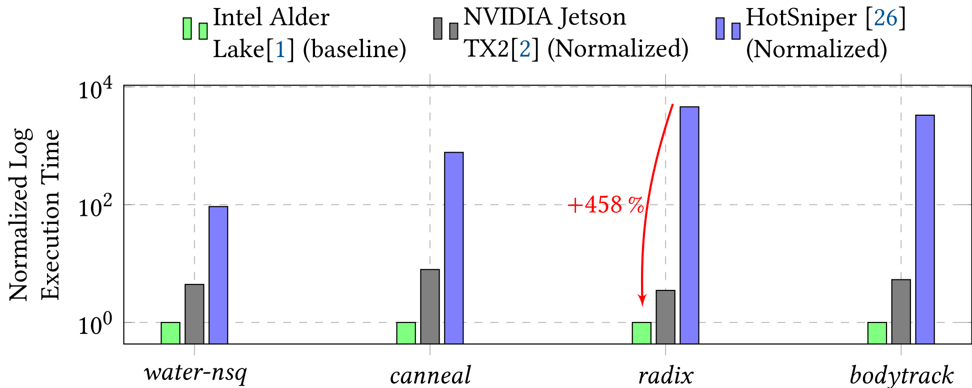


Fig. 1. Despite the benefits of simulators for architectural exploration, they often exhibit significantly longer execution times compared to real hardware. This figure demonstrates the normalized execution times of various benchmark applications, highlighting the drastic slowdown in the HotSniper[26] simulator, which can be several orders of magnitude higher than real hardware such as Intel Alder Lake and NVIDIA Jetson TX2. Such long simulation times can be a major bottleneck in exploring system-level resource management.

These efforts have demonstrated key benefits of hardware-based RM experimentation, including improved accuracy, reduced experimentation overhead, and applicability to real-world scenarios.

Despite these advantages, using real hardware for RM research comes with significant drawbacks. Current hardware platforms are highly heterogeneous and lack a unified framework for experimentation. Consequently, researchers are often forced to "reinvent the wheel," developing *ad hoc* tools and methodologies for their approaches. Additionally, these implementations are typically *not portable* across architectures, limiting their *reproducibility* and *scalability*. These challenges hinder the broader adoption of real-hardware-based approaches in RM research.

To address these limitations, we introduce ARDiS, the first open-source[2] *portable* framework for system-level resource management on real hardware platforms. ARDiS is designed to overcome the drawbacks of existing hardware-based RM research and integrate the flexibity and reconfigurability of simulation frameworks, by providing a unified, portable, extensible and architecture-agnostic platform. Specifically, ARDiS enables researchers to implement and evaluate sophisticated RM techniques, including ML-based strategies, with minimal overhead, achieve high accuracy in performance evaluation by leveraging real hardware behavior, facilitate reproducibility and portability across different hardware architectures, and reduce development time by providing a robust and reusable infrastructure.

Considering the above, in this article, we present the following novel contributions:

- We propose ARDiS: A portable, unified, and open-source resource management framework aimed at standardizing and evaluating RM policies on real hardware platforms.
- Through extensive experimentation on two distinct hardware platforms, we present a functional evaluation of ARDiS and its submodules, demonstrating its fine-grained monitoring and RM policy integration capabilities, as well as its ability to generate experimental data suitable for training ML models.
- We present two use cases where we replicate, run, and evaluate state-of-the-art techniques within our framework, showing its flexibility and ease of use in porting RM techniques from simulation to real hardware platforms.

[2]https://github.com/Chair-for-Embedded-Systems/ARDiS

2 Design Principles, Requirements, and Challenges

To effectively address existing gaps in the literature and practical limitations inherent in real hardware-based RM, our proposed framework must adhere to carefully defined design principles and face key technical challenges. The subsequent subsections outline the fundamental design principles and requirements that guide our approach, and detail critical implementation challenges, emphasizing their significance and the solutions we adopt to overcome them.

2.1 Design Principles and Requirements

In developing a robust and effective framework to bridge existing literature gaps and overcome practical limitations inherent in real hardware-based RM, we outline fundamental design principles and requirements essential to guide our approach. (1) Portability, to be hardware-agnostic and to ensure compatibility across different architectures and platforms. (2) Comprehensive RM support, to provide mechanisms to support multiple RM objectives and control knobs. (3) Facilitating training data generation, to efficiently collect high-quality training data for ML-based RM approaches. Finally, (4) ease of use and extensibility, to be modular, open-source, and designed for easy extension, to encourage widespread adoption in the research community.

To achieve (1) portability, we develop a hardware abstraction layer that provides an interface for interacting with different target architectures. This abstraction ensures that RM policies can be implemented independently of the underlying hardware constraints. The architecture of our framework is also layered, with a clear separation of concerns between its different decoupled modules. For (2) comprehensive RM support, the framework is structured around the primary RM control modules: scheduling, DVFS management, application mapping, and application migration. These modules are configurable components, allowing researchers to define, modify, and extend RM strategies in a unified environment. To address (3) training data generation, we implement a flexible and parameterized monitoring module. It allows the users to specify the level of monitoring (per-core, per-process, system-wide), the set of parameters to be logged (execution time, power consumption, cache utilization, etc.), as well as the granularity of data collection. In addition, the monitoring module serves real-time statistics in a continuously-updated parametrized buffer that can be used following the periodic invocations of the resource management policies, e.g, to be passed as input to a machine learning model for inference. Finally, for (4) ease of use and extensibility, the framework adopts a fully modular design, implemented in *Python* to maximize accessibility. Each module is parameterized, and low-level OS-specific functionalities are encapsulated in high-level functions, further facilitating portability and maintainability. Furthermore, by making the framework open-source, we encourage adoption, collaboration, and continuous improvement.

2.2 Design Challenges

Despite its modular and configurable design, several technical challenges arise in developing ARDiS.

Challenge 1: Generalization vs. Complexity. A fundamental challenge is determining the appropriate level of generalization for RM functionalities. The framework must be broad enough to support diverse RM techniques while avoiding excessive code complexity and unnecessary interfaces. To address this, we surveyed a wide range of RM techniques in the literature and designed our framework to ensure compatibility with established methodologies.

Challenge 2: Synchronization Across RM Knobs. Sophisticated RM strategies often require multiple control knobs (e.g., dynamic frequency scaling combined with workload migration) to be applied concurrently, which introduces challenges in synchronization. Without proper coordination, concurrent modifications to shared system resources, such as core frequency settings or thread affinity mappings, may lead to race conditions, inconsistent states, or performance degradation. To

ensure coherence, we implement an epoch-based synchronization mechanism, where all RM policies operate within predefined execution windows. A global synchronization signal ensures that policies apply their updates only at well-defined epochs, preventing uncontrolled interference. As shown in Figure 3, our monitoring and reporting module rely on *Queues* to serve the monitored data to the different in parallel to the different consumers. This ensures that policies do not interfere with each other and that execution follows a predefined priority ordering. Additionally, *Read-Write Locks* allow multiple policies to read system state concurrently while ensuring exclusive access during updates. To avoid conflicts when policies execute in separate threads, *Thread Locks* enforce mutual exclusion when modifying shared variables, such as per-core frequency levels or application mappings. This combination of epoch-based coordination and fine-grained locking ensures that multi-knob RM techniques operate efficiently while maintaining synchronization across concurrent control decisions.

Challenge 3: Lightweight Implementation. To be practical for real-world use, the framework must introduce minimal overhead. If RM enforcement itself consumes excessive system resources, it will interfere with the applications being managed. To mitigate this, we optimize the implementation by leveraging Python's efficient multi-threading to reduce blocking operations, minimizing contention in RM policy execution. Finally, the framework is structured to operate at *user-level*, avoiding *kernel*-space modifications that would introduce additional overhead.

3 ARDiS: Our Portable and Unified Resource Management Framework

We present ARDiS, the first open-source and portable framework that provides a unified, architecture-agnostic platform for running system-level resource management techniques directly on real hardware. An overview of ARDiS is illustrated in Figure 2. ARDiS is designed as a modular and flexible framework that enables system-level resource management techniques to be executed in a unified, architecture-agnostic manner on real hardware. The following subsections provide a detailed breakdown of the design requirements and challenges of ARDiS, before describing the execution workflow of the framework and explaining the role and implementation details of each of its components.

3.1 Resource Management Policies

ARDiS supports all main system-level resource management knobs: *scheduling*, application/thread *mapping*, application/thread *migration*, and *DVFS*. Users of ARDiS can choose from default policy templates provided by the framework or define custom policies for each of these RM knobs.

The *Scheduling Policy* module manages the execution order and arrival times of applications on available cores. It ships with various predefined strategies, including unified arrival times, static consecutive schedules, and dynamic random scheduling based on probabilistic distributions.

The *Mapping Policy* determines how applications and threads are assigned to cores. It supports two main strategies: *static mapping*, in which threads are assigned to specific cores based on predefined rules and remain there throughout their execution, and *dynamic mapping*, which specifies only the initial cores for threads at application startup. To further refine thread assignments during runtime, the *Migration Policy* dynamically alters thread-core affinity based on system conditions or user-defined criteria.

The *DVFS Policy* allows adjusting the core **Voltage and Frequency (VF)** levels to optimize performance and energy consumption. This policy supports multiple approaches, including static frequency assignment or reactive/proactive scaling driven by current workload demands, e.g., ML driven. Additionally, this module can directly utilize default Linux governors such as *performance*, *schedutil*, *powersave*, and so on. DVFS policies make use of the built-in *Translator* utilities, which map user-specified VF levels to hardware-specific representations such as Intel's P-states. These default templates serve as baseline strategies for RM research and can be flexibly synchronized and combined to investigate sophisticated multi-knob RM techniques.

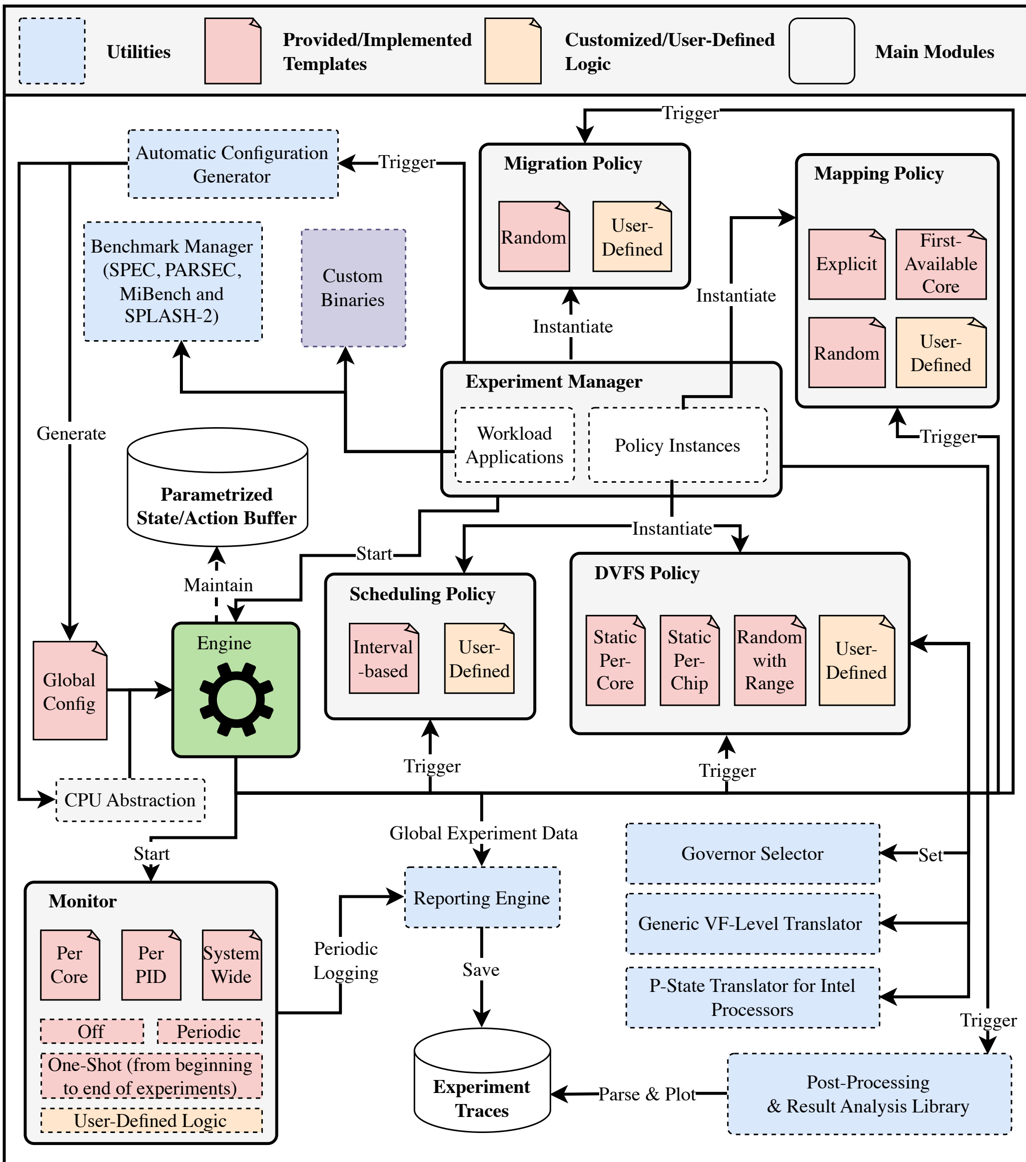


Fig. 2. High-level overview of ARDiS's architecture and modules.

3.2 Preparing for Experimentation

The **Experiment Manager** module in ARDiS is responsible for setting up and defining the parameters for each experiment conducted within the framework. It is conceived to be the main entry point for ARDiS users to quickly kickstart their RM experiments. To start an experiment, the user shall specify the applications to run, along with instantiations of different policies for scheduling of the applications, mapping them to cores, and setting the VF levels of their assigned cores. In addition, the user should also specify whether or not they would like to migrate applications throughout the experiment, i.e., affinity updates.

Before starting an experiment, the user shall also prepare the **Global Configuration**. It is a configuration file that is hardware- and experiment-specific to specify many parameters. To quickly kickstart experimentation, our **Automatic Configuration Generator** can be used to generate a preliminary configuration file that is adapted to the capabilities of the system, i.e., number of cores, available sensors, available **performance monitoring counters (PMCs)**, default monitoring epoch, default DVFS epoch, and so on. Based on such a base configuration, the user can adjust the parameters to the specific needs of the intended experimentation. This approach ensures that each experiment is reproducible and that the results are comparable across different runs.

3.3 The ARDiS Workflow

ARDiS's workflow is detailed in Figure 2. The experiment manager is responsible for starting the **Engine**, the *central* component that orchestrates the execution of the experiments. Based on the parameters specified in the global configuration file and the different instantiations of the RM policies, i.e., scheduling, mapping, DVFS and migration, the engine effectively starts the experiment and the supporting mechanisms of benchmark management, binary execution, monitoring, reporting, and so on. If the experimentation workloads are benchmark applications from *PARSEC*, *SPLASH-2*, *SPEC*, or *MiBench*, the engine invokes their corresponding benchmark managers through the **Benchmark Manager** utility.

At the beginning of each experiment, the engine starts the **Monitor** module to collect performance data during the execution of applications, based on the granularity and PMCs provided in the global configuration. The monitored data can include metrics such as execution time, power consumption, and a diversity of CPU and cache events, using *perf* [3]. The data is collected in two modes: *one-shot* and *periodical*. One-shot monitoring starts the *perf* thread at the beginning of the execution and stops it when the experiment finishes. This is useful when only the total value of a metric is needed at the end of the experiment, e.g., total energy consumption. The monitored one-shot metrics are saved at the end of the experiment to an *execution summary* file. On the other hand, per-core, per-PID, and system-wide monitoring are periodical by definition. The monitored metrics in this mode are saved periodically to a *periodic log* file. As ARDiS supports the execution of multiple applications/threads in parallel, and the monitoring of multiple metrics periodically, recording these metrics *sequentially* and *synchronously* can lead to significant overheads that exceed the epoch time window. To tackle this, we implement parallelization and queuing mechanisms between the monitor and the *Reporting Engine*, to manage the different invocations of *perf* at runtime, as shown in Figure 3. This robust synchronization contributes significantly to the fulfillment of the low overhead requirement in Section 2, and as will be demonstrated in Section 4, also enables users to develop complex resource management strategies that combine multiple knobs, e.g., migration and DVFS.

The engine also integrates the periodic invocation of the various policies and the collection of execution metrics from the monitor module to execute the experiments according to the defined configuration and instantiated policies. To support the decisions of each individual policy, the user can configure **Parametrized State/Action Buffers**, which serve the monitored metrics to the policies directly from memory without having to access the execution traces at runtime. After creating threads for each application in the experimentation workload, the engine periodically tracks their states, PIDs, core mappings, and their VF levels in a synchronized and thread-safe manner that prevents race conditions for the different shared *state data* at runtime. Examples of such shared *state data* are the PIDs of applications, thread-to-core-mappings, and VF levels of cores, which can be accessed and altered by different policies simultaneously.

It is important to note that ARDiS's engine maintains in the buffers the *intended* decisions of the RM policies along with their *actual* impact on the system, to prove that RM decisions are *effectively* applied on the operating system. For instance, if a migration policy *intends* to alter the

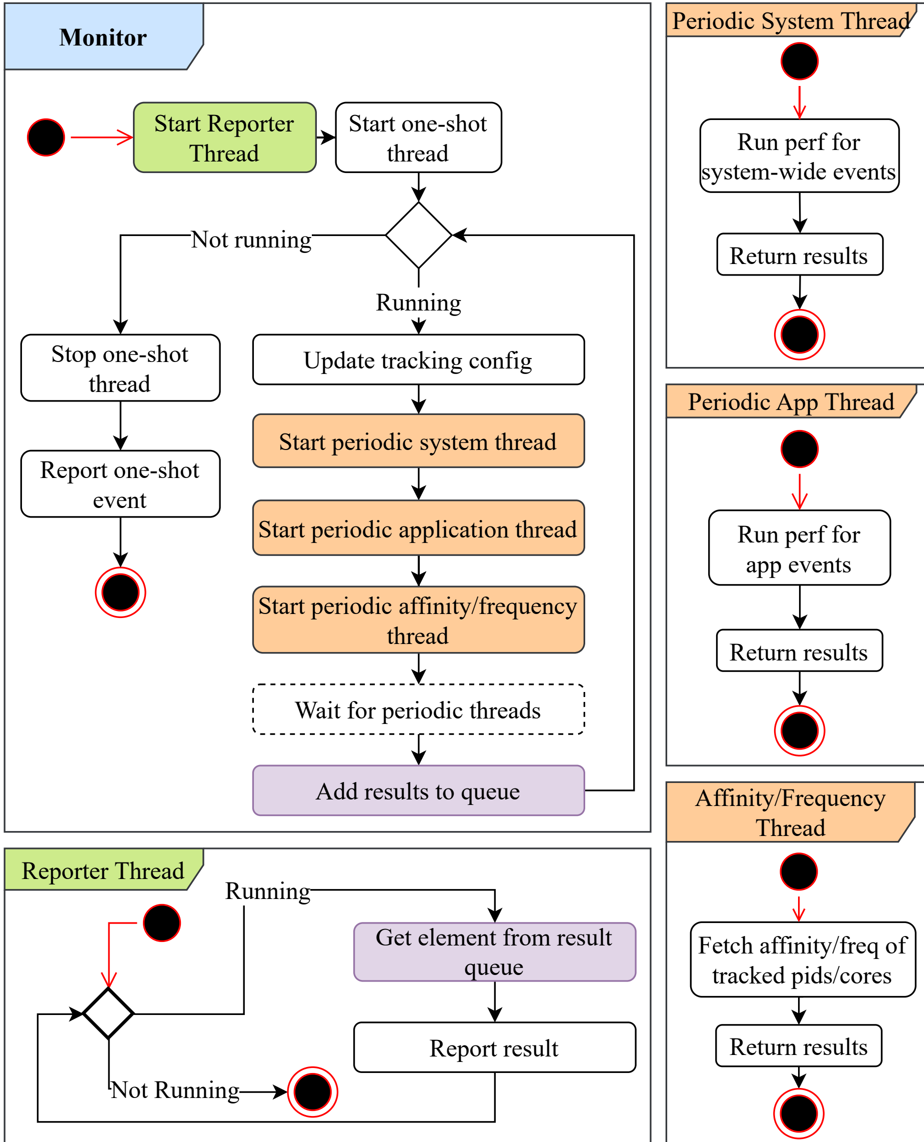


Fig. 3. Overview of ARDiS's multi-threaded monitoring system, where periodic and one-shot monitoring threads coordinate to track system-wide events, application-specific metrics, and affinity/frequency updates, ensuring efficient, low-overhead, and synchronized data collection at runtime.

thread-to-core mapping of an application, or if a DVFS policy *intends* to boost the VF level of a core, the affinity/frequency monitoring thread will track the *actual* affinity of the corresponding process and the VF level of the corresponding core, respectively.

Finally, once the workload has finished execution, and to ensure reproducibility across experiments, the engine clears the caches. This is crucial for characterization experiments that require

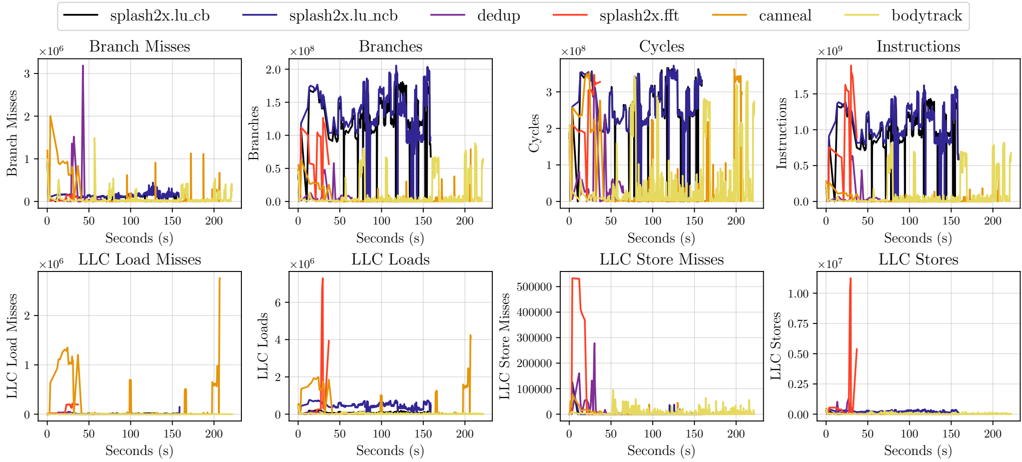


Fig. 4. The result analysis library is triggered automatically at the end of each experiment to post-processes its periodic execution logs and to generate plots. This feature is configurable, and by default generates one plot for each monitored metric in ARDiS, thereby enabling researchers to easily analyze the results of their experiments out of the box.

repeated runs of the same experiment; a warmed-up cache could introduce misleading experimentation noise in the performance traces. The engine is also responsible for stopping the monitor and for triggering the post-experiment-processing, as detailed in the following subsection.

3.4 Post-processing and the Result Analysis Library

In a consumer/producer design pattern, the reporting engine saves the experiment traces at the end of the execution of the last application in the workload, while the *result analysis library* post-processes this data to generate analysis plots and diagrams in the experiment result folder. As shown in Figure 4, the post-processor automatically—unless configured otherwise, generates one plot per monitored *perf* event/metric and frequency over time. Additionally, it produces a mapping plot that visualizes the cores on which applications—and their respective threads—have executed over time, as illustrated in Figures 5 and 6. This feature is particularly useful in scenarios where a migration policy is enabled, as it helps verify the correctness of the thread-to-core affinity settings over time.

Furthermore, depending on the sensing capabilities of the system, the post-processor can also generate power/energy plots. Similar to other modules of ARDiS, the post-processing functionality is fully customizable, allowing users to specify which plots to generate for each experiment. Moreover, it provides the option to selectively plot the aforementioned visualizations for specific *applications of interest* within the workload, which can be particularly useful for reducing plot complexity and improving readability when analyzing large-scale workloads.

3.5 Support for Multi-threaded Workloads

As highlighted in Figure 2, ARDiS supports the execution of different benchmark applications, e.g., PARSEC [4], SPLASH-2 [35], SPEC [17], and so on. As most of these benchmarks support multi-threaded execution, ARDiS's monitoring capabilities enable researchers to track all spawned threads individually during an execution, but also allow the grouping of their individual traces into application-level plots at the end of an experiment. Such capabilities are highlighted in the plots in

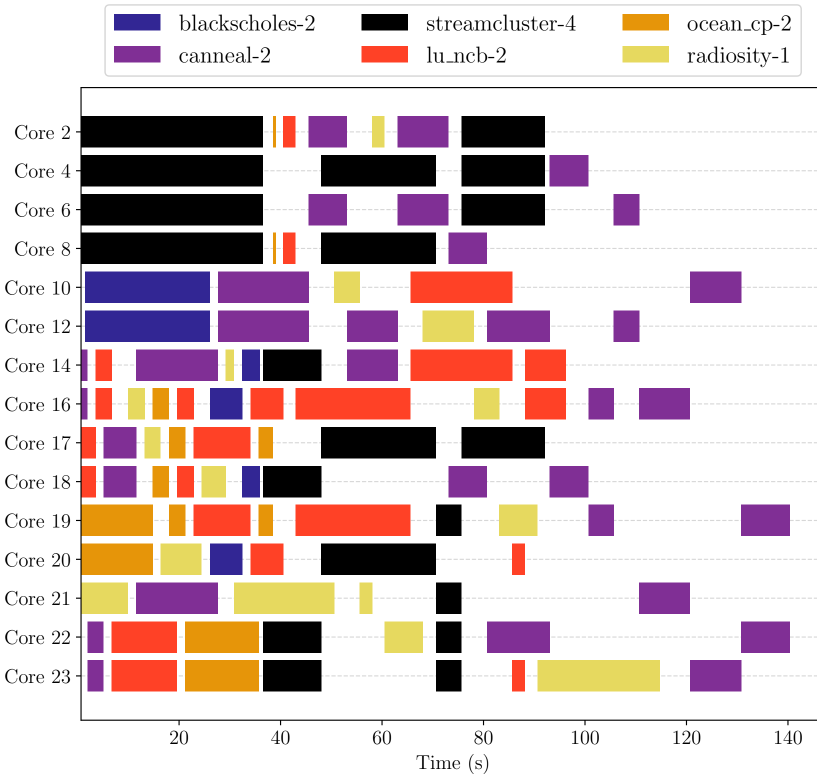


Fig. 5. Multi-threaded PARSEC and SPLASH-2 benchmark applications executed with a dynamic migration policy. ARDiS's monitor is capable of tracking all children threads of each application process. This fine-grained collected allow ARDiS to generate a thread-to-core affinity tracking map of all executed threads, organized per workload application.

Figure 7, where the results of an experiment on ARDiS can enable the analysis of scaling behaviors of different applications given their inherent characteristics, e.g., **thread-level parallelism (TLP)**, instruction-level parallelism, compute intensity, and so on. Moreover, as can be observed in Figure 8, some benchmark applications can spawn more threads than originally expected, e.g., 20 threads instead of the specified 4. Such scenarios require the monitoring framework to be able to capture the PID of any spawned thread by any running application, regardless of the *expected* number of threads. This is a crucial feature for RM research, as failure to capture all children PIDs in such cases can lead to drastically lower PMC readings, e.g., missing the PMC readings of 16 threads out of 20.

4 Experimental Evaluation

In this section, we start by presenting an extensive functional evaluation of the different modules of our framework. Second, we evaluate our ARDiS on use cases from different domains in the literature where RM techniques are employed, by porting them from their original simulation-based implementations to real hardware. Third, we compare ARDiS against the state-of-the-art framework LUSH [36] both functionally and experimentally. Finally, we evaluate the runtime overhead of ARDiS when executing large multi-application workloads on two different real target platforms.

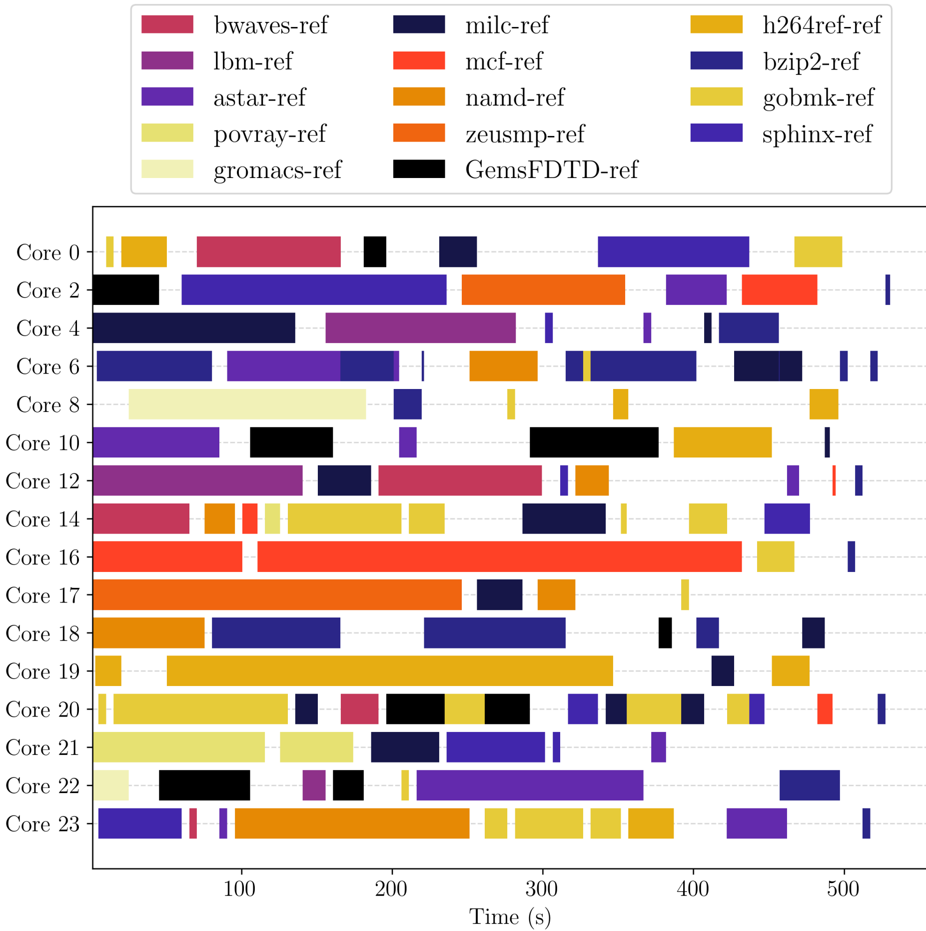


Fig. 6. Single-threaded SPEC CPU2006 benchmark applications executed with a dynamic migration policy. After monitoring their execution throughout the experiment, ARDiS generates a thread-to-core affinity tracking map of all executed applications in the workload. This feature is especially important for verifying the correctness of affinity settings of applications over time, when task migration policies are enabled in ARDiS.

4.1 Experimental Setup

To evaluate our framework on varied and heterogeneous computing platforms, we selected two main setups. The first one is a server-grade Linux-based machine, equipped with the heterogeneous Intel® Core™ i9 12th generation Alder Lake heterogeneous processor, with eight performance cores (P-cores) and eight efficiency cores (E-cores). This platform represents an example of a server-like architecture for multi-core resource management scenarios.

The second platform is the NVIDIA Jetson TX2 board. This board is built on a heterogeneous architecture featuring two distinct clusters. One cluster houses a Quad-Core ARM Cortex-A57, while the second contains a Dual-Core NVIDIA Denver 2 64-bit CPU. This platform represents a heterogeneous embedded computing scenario, composed of clusters of cores with different capabilities that follow the trend of modern high-end embedded devices such as those in the automotive or mobile industry.

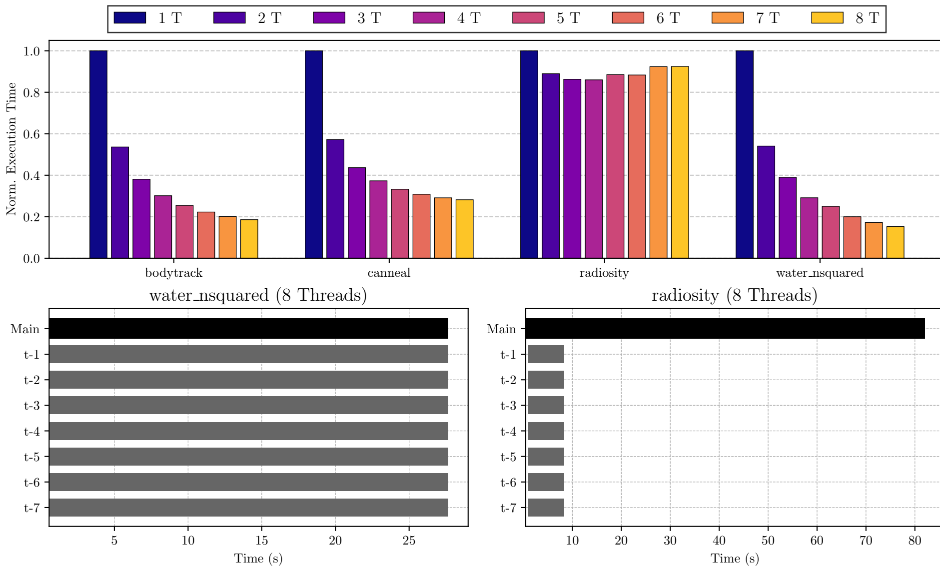


Fig. 7. Selected applications from the PARSEC and SPLASH-2 benchmark suites, executed at different supported parallelism levels, from one to eight threads, executed at a fixed VF level. The experiment highlights that different applications benefit differently from work parallelization. While the 8-threaded *water.nsquared* exhibits more than 80% performance increase compared to its single-threaded version, *radiosity*'s performance increases by barely 10% at higher parallelism levels. This behavior is explained by TLP of each application. For *water.nsquared* all threads actively perform the task in parallel throughout the execution of the application, demonstrating a higher TLP compared to *radiosity*, where 7 out of 8 threads are only active for the first 8 seconds of the 83-second execution of the application.

4.2 Functional Evaluation

First, we run characterization experiments of the *PARSEC* and *SPLASH-2* benchmark suites on the heterogeneous Intel® Core™ i9 12th generation Alder Lake processor. Each application is executed on the system in the seven modes described in Table 1. These experiments intend to test the main modules of ARDiS: the *Engine*, the *Monitor*, DVFS, mapping and migration policies, *Reporting Engine*, and the *PARSEC* and *SPLASH-2 Benchmark Managers*. The *Mixed* data are results from running an energy-aware technique that combines task migration with DVFS with the goal of minimizing energy consumption. Depending on the execution phase of the application, the technique selects the core and the VF level that offers the lowest energy consumption. To know when to migrate, the migration policy leverages an execution mapping schedule constructed by profiling the applications offline. The obtained traces at the end of each experiment are post-processed to extract performance and energy plots. Since the Intel processor is not equipped with power sensors, the *RAPL* interface is used to obtain energy measurements.

The results shown in Figure 9 confirm the expected performance and energy trends of the different executions, confirming the correctness of different aspects of the ARDiS workflow. Specifically, the *P-core max* (at 3.5 GHz) execution shows a performance comparable to the *performance* governor, which also selects the maximum VF level. Alternating between P-cores and E-cores in the *Mixed* experiments indeed leads to a worse performance for all applications, but also a lower energy consumption for many of them. Finally, executions with the *powersave* governor lead to the highest energy consumption and the worst performance for all applications. This is expected, as the *powersave* governor aims at minimizing the absolute periodic power consumption by throttling the

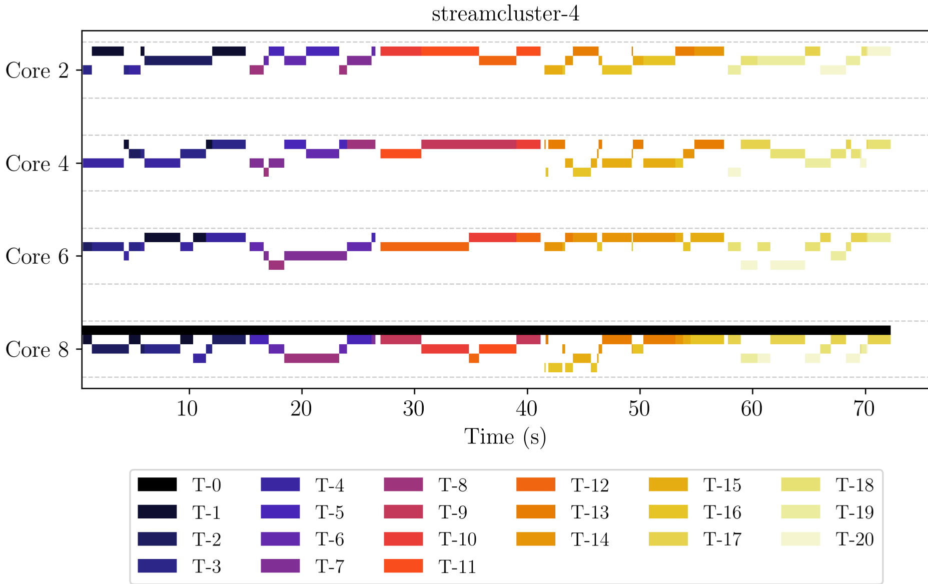


Fig. 8. A detailed per-thread trace of *streamcluster* from the PARSEC benchmark suite, executed with parallelism level 4. The per-thread mapping trace reveals that *streamcluster* spawns a total of 20 threads, instead of the expected 4 threads, thereby demonstrating the adaptiveness and robustness of ARDiS in dealing with scenarios of unpredictable thread numbers, where the *expected* number of reported threads according to official documentations of the benchmark suites is not accurate.

Table 1. The Different Experiment Setups Used in the Evaluation Experiments in Figure 9

Experiment	Description
P-core at 3.5GHz	From start to finish on a P-core
Mixed	Energy-aware DVFS and migrations
Performance	Linux's *performance* governor
Powersave	Linux's *powersave* governor
Ondemand	Linux's *ondemand* governor
Conservative	Linux's *conservative* governor
Schedutil	Linux's *schedutil* governor

cores to the lowest VF level, thereby significantly extending the execution time of applications and leading to a higher energy consumption.

4.3 Machine Learning Integration and Porting Capabilities of ARDiS

As discussed in Section 1, ML-based resource management techniques are becoming the norm in the literature. The goal of the following experiment is to evaluate the capabilities of ARDiS in generating training data for training ML models, that can enable the development of smart resource management techniques.

We run experiments using the *PARSEC* and *SPLASH-2* benchmark suites. Each application is executed on both a P-core and an E-core on the heterogeneous Intel® Core™ i9 12th generation Alder Lake processor at VF levels from 1.5 GHz to 3.5 GHz in 100 MHz increments. In each experiment,

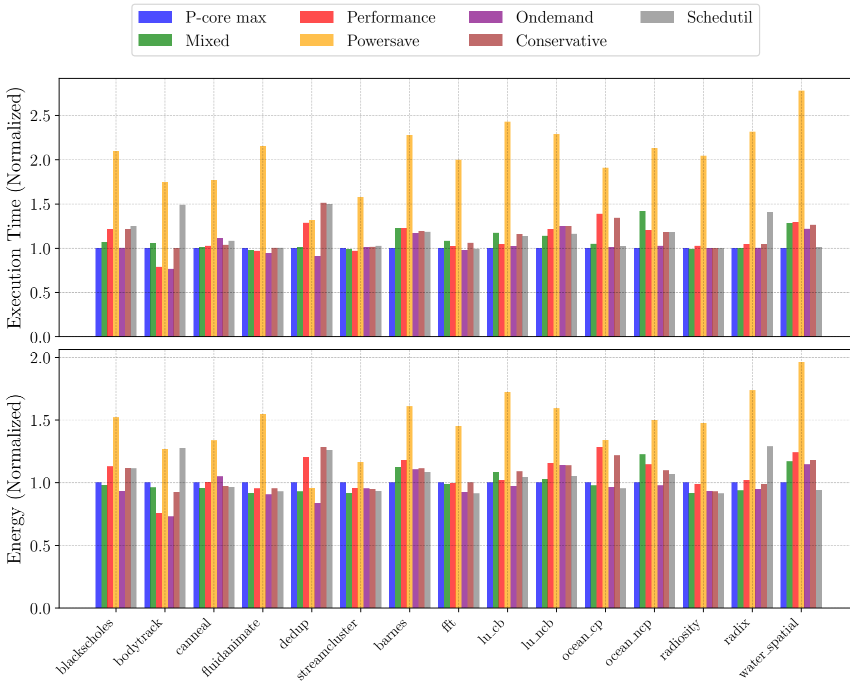


Fig. 9. Normalized performance and energy measurements of different benchmark applications, when running with different strategies on ARDiS. In addition to the default governor and maximum VF level executions, we also present the *Mixed* execution, which evaluates a combined DVFS and migration RM technique that periodically adjusts both the VF level and the core affinity of running applications to minimize energy consumption.

PMCs are collected every 100 ms, tracking retired instructions, cache accesses/misses, cycles, and branches/misses. Energy consumption is monitored using the RAPL interface at the same granularity. The obtained experiment traces are sliced similarly to [28] to match similar phases of the same applications when executed on each core type at each VF level. We characterize each slice by all the recorded PMCs as features, and with the energy consumption in that slice as a label.

All slices are used to construct a dataset of 250,000 samples. We then train different ML models to predict the energy efficiency, computed as **instructions per Joule** (**IPJ**), given the recorded features in each slice at a specific configuration of core type and VF level. We experimented with three machine learning models: XGBoost Regressor, Neural Network, and Random Forest Regressor, trained with an 80–20 training-validation split of the obtained training dataset. We introduced meta-features such as **instructions per second (IPS)** and **instructions per cycle (IPC)**, and scaled all features using a min-max scaler. The XGBoost Regressor was trained with the default hyperparameters. The Neural Network used a sequential architecture with two hidden dense layers (128 and 64 neurons), optimized by Adam and MSE loss, with early stopping. The Random Forest Regressor was implemented with 100 trees and a random state of 42.

To comprehensively assess the performance of our ML models and the accuracy of data collected through ARDiS, we employ several standard regression metrics. **Mean Absolute Error** (**MAE**) and **Root Mean Squared Error** (**RMSE**) quantify the average magnitude of prediction errors, with RMSE being more sensitive to outliers. The R^2 Score (Coefficient of Determination) indicates the proportion of variance in the dependent variable explained by the model, with values closer

Table 2. Comparison of the Performance of Different ML Models on the ML Task from the Literature [28]

Model	MAE	RMSE	R^2 Score	MAPE
Random Forest	0.0712	0.1624	0.9843	4.29%
Neural Network	0.1066	0.1926	0.9779	6.80%
XGBoost	0.1058	0.1935	0.9777	6.67%

to 1 indicating a better fit. **Mean Absolute Percentage Error** (**MAPE**) expresses prediction accuracy as a percentage, providing an intuitive understanding of relative error magnitude. Together, these metrics provide complementary perspectives on model performance: MAE and RMSE assess absolute prediction errors, R^2 evaluates overall model fit, and MAPE enables interpretation in terms of percentage deviation from actual values.

The models were evaluated using the metrics defined above, with results presented in Table 2. **Model Performance Analysis:** All three models demonstrate strong predictive capabilities, achieving R^2 scores exceeding 0.97, which indicates that over 97% of the variance in IPJ values is explained by the input features. The Random Forest model emerges as the top performer with an MAE of 0.0712 and MAPE of 4.29%, meaning predictions deviate from actual values by less than 5% on average. The Neural Network and XGBoost models exhibit comparable performance (MAE of 0.1066 and 0.1058, respectively), with MAPE values around 6.7%, which is still excellent for energy efficiency prediction tasks.

Framework Validation: The consistently high performance across all three models provides strong evidence for several critical aspects of ARDiS. First, the low error rates (MAPE < 7% for all models) indicate that data collected through ARDiS exhibits minimal noise, demonstrating effective isolation of system-level interference and accurate capture of performance and energy metrics at the hardware level. Second, the high R^2 scores (> 0.97) confirm that the performance counters and system metrics exposed by ARDiS contain sufficient information to accurately predict energy efficiency across different configurations, validating our choice of monitored features. Third, and most significantly, these results demonstrate that the execution trace slicing methodology, originally validated in simulation [28], translates seamlessly to real hardware through ARDiS. The comparable accuracy to the original simulation-based work confirms that ARDiS successfully bridges the simulation-to-hardware gap without sacrificing data fidelity or prediction quality. Finally, the narrow performance gap between different ML architectures (Random Forest, Neural Network, and XGBoost) suggests the problem is inherently learnable and well-characterized by the collected features, rather than being dependent on specific model selection or extensive hyperparameter tuning. This consistency across models and the low error rates validate ARDiS's capability to provide reliable, reproducible data suitable for training sophisticated ML-based resource management techniques on real hardware platforms.

Generalization Analysis: As a common evaluation step for machine learning models, we also evaluate their ability to generalize to unseen applications during training. We performed a cross-validation experiment with the XGBoost Regressor, where specific applications were excluded from the training set in each fold, while the test set consistently included all applications. This setup simulates a scenario where the model must predict performance for applications it has not been trained on but has access to data from other applications, which is crucial for the generalization of ML-based resource management techniques at runtime. We used the GroupKFold cross-validation method to divide the dataset, ensuring that each fold excluded one application from training while the entire dataset was used for testing. In each fold, performance was evaluated using the same metrics. The detailed results for each fold are summarized in Table 3.

Table 3. Cross-validation Results for Application-specific Exclusion Experiment

Fold	Excl. App	MAE	RMSE	R^2	MAPE
1	*blackscholes*	0.12	0.20	0.98	6.94%
2	*bodytrack*	0.13	0.24	0.96	7.54%
3	*dedup*	0.11	0.20	0.98	8.98%
4	*streamcluster*	0.17	0.28	0.95	10.15%
5	*fft*	0.12	0.26	0.96	6.93%
6	*lu_cb*	0.12	0.20	0.98	6.93%

Each fold excludes a different application from training while using the entire dataset for testing.

The results indicate that the model achieves strong predictive performance even when certain applications are excluded from the training set, maintaining a ∼ 7% MAPE across most of the folds, a very comparable error to the version trained with data from all applications Table 2.

4.4 Security Use Case

In order to further evaluate ARDiS integration capabilities, we implemented a use case where different RM knobs are used in the security domain. For this purpose, we chose to replicate the work from [19] (TCAD'21). In their work, the authors present a **Discrete Fourier Transform (DFT)** based detection technique and a DVFS countermeasure mechanism for power-based covert-channel attacks (e.g., **thermal covert channels** (**TCCs**)). This work was originally evaluated using the Sniper simulator [7] in a many-core setup. To highlight its flexibility and integration capabilities, in our use case, we port their implementation to ARDiS to evaluate their approach on the NVIDIA Jetson TX2 platform. As an embedded board with a heterogeneous multi-core architecture, this platform represents a different and interesting scenario to validate the results from [19].

4.4.1 Effectiveness of ARDiS-based Attacks. In our first experiment in this use case, we implement the power-based covert channel with both transmitter and receiver functionalities. For the transmitter, we employ a return-to-zero line encoding and an **on-off-keying** (**OOK**) modulation scheme. As a proof of concept for the attack, we set a transmission frequency of approximately 15 Hz. We execute the covert channel transmitter application in ARDiS, sending 800 bits, encoded as 8-bit packets, achieving a transmission rate post modulation of approximately 2 bps. While the attack runs, we collect the traces from the power sensor of the Jetson board periodically by leveraging ARDiS's monitoring capability. By applying a simple demodulation on the power signal over time, we are able to decode the packets with an average **bit error rate (BER)** of 4.6%. This shows that, though achieving a reduced transmission rate on a real system compared to the simulation-based results reported in [19], implementing such an attack on real hardware platforms is possible and easy using ARDiS.

4.4.2 DFT-based Detection. The next step we followed in replicating the original work is the detection technique. Although some of the implementation details are missing from the original work, we followed the proposed threshold-based detection approach on the magnitude of the DFT of the core performance. In our implementation, we take samples from each core's performance every 10 ms using ARDiS. Then we form windows of data of 500 samples. After taking the samples, we apply a zero-padding scheme to extend the window to 2,048 samples before applying the DFT. Following this method, we collect traces from normal system operation (i.e., benchmark application execution from the SPEC CPU 2006 suite [17]) and while running the TCC attack. For

Table 4. Performance Metrics for the TCC Detection Techniques

Technique	Dataset Size	Platform	Accuracy	FNR	FPR	F1 Score
TCAD'21 [19]	not reported	Sniper (2D and 3D architectures)	97.0%	not reported	not reported	not reported
ARDiS implementation	~100,000 samples 50% attack 50% benchmark	Jetson TX2	92.5%	0.02	0.13	0.93

this experiment, we collected a total of 100,000 samples, evenly split into attack and normal traces. After the dataset is formed, we apply the heuristic detailed in [19] to find an optimal threshold value to classify between normal and attack traces.

Table 4 shows the resulting metrics from our implementation of the work from [19] using our ARDiS framework on the Jetson TX2. Beyond overall accuracy, we evaluate the **False Negative Rate** (**FNR**), which measures the proportion of attacks incorrectly classified as benign; the **False Positive Rate** (**FPR**), indicating benign workloads incorrectly flagged as attacks; and the F1 Score, which provides a balanced measure combining precision and recall.

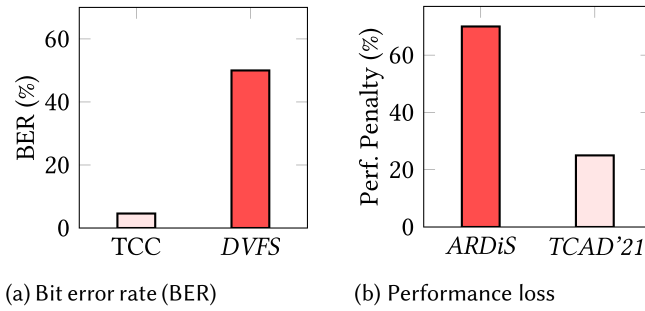
As seen in the table, our replication managed to achieve a detection accuracy of 92.5%. While slightly lower than the value reported in the original work, this result shows that the technique indeed is successful in detecting attacks in real systems. In particular, our implementation of the approach has a very low FNR, which means that most of the attack samples are classified as such.

4.4.3 DVFS Countermeasure. As suggested in the original work [19], the default countermeasure to the TCC is DVFS. By dynamically switching a core from a high frequency to a low frequency at a certain down-up date (β), power noise is generated and the channel gets jammed.

To evaluate the effect of the DVFS countermeasure, we implemented the technique as a DVFS policy in ARDiS, using $\beta = 9$. As in their work, we employ the maximum frequency of the board as the high value, and a random choosing of the four lowest frequencies available as the low value. In this experiment, we evaluate the countermeasure in two manners. First, we evaluate the effectiveness of the policy at mitigating the attack. For this, apply the DVFS policy every 1 second in ARDiS, while the transmission of the 800 bits is active. Figure 10(a) shows the BER of the TCC with and without the countermeasures. As seen, our replication of the original technique is able to induce a very high error rate in the transmission, effectively nullifying the attack. This experiment confirms that our ARDiS implementation of the DVFS technique works in the new platform within our framework.

Finally, we measure the performance penalty in the system due to the DVFS countermeasure. In this experiment, we ran 50 random workloads consisting of five applications from the SPEC benchmark and the TCC transmitter, with one application executing per core. We run all workloads using ARDiS twice: first with the DVFS policy disabled and then with the countermeasure policy active. Figure 10(b) shows performance penalty due to the countermeasure both in the original work and in our implementation. As can be seen, the penalty due to the countermeasure is approximately 70% in ARDiS, which greatly differs from the results obtained in [19] where a performance loss of approximately 25% is reported under the same attack scenario, i.e., when the attacker is present at all times.

Notably, this high performance loss does not come from our framework, as the overhead introduced by the periodic monitoring is minimal, as further discussed in Section 4.7. We believe that this is one of the most interesting results from the use case. As previously discussed, replicating the approach on a different, and more importantly, real-world platform introduces new variables which



(a) Bit error rate (BER) (b) Performance loss

Fig. 10. (a) BER for the baseline TCC (no countermeasure) and while the DVFS policy is active. (b) Average performance penalty (loss) due to the DVFS countermeasure when the attack is present at all times from the original work (TCAD'21) and ARDiS's implementation.

are not present in a simulation scenario. One of the key differences in the reported overheads is the underlying hardware. As the Jetson TX2 platform has a shared DVFS domain per cluster, while active, the countermeasure affects all the CPUs within the cluster where the potential transmitter executes. By facilitating the integration of state-of-the-art resource management techniques into a controlled execution framework on real hardware, ARDiS allows researchers to validate existing work and possibly find previously unobserved phenomena, such as this one, that could help expand the research field.

4.5 Simulation vs. Real Hardware: Training Data Generation Time

To quantitatively demonstrate the practical advantages of ARDiS over simulation-based approaches, we compare the time required to generate training data for the ML-based resource management technique discussed in Section 4.3. This comparison directly addresses the simulation bottleneck discussed in Section 1 and illustrated in Figure 1.

4.5.1 Experimental Setup. As discussed in Section 4.3, we generate a comprehensive training dataset by executing all applications from the PARSEC and SPLASH-2 benchmark suites across multiple system configurations on the Intel® Core™ i9-12900K processor. Specifically, each application was run on both P-cores and E-cores at 21 different VF levels, ranging from 1.5 GHz to 3.5 GHz in 100 MHz increments. PMCs and energy consumption metrics were sampled every 100 ms throughout execution. For the simulation-based comparison, we replicated this exact experimental configuration using HotSniper [26], as in the original work [28]. The simulations were distributed across four compute nodes: one AMD Ryzen 9 3900X (12-core, 4.6 GHz) and three AMD Ryzen 7 2700X processors (8-core, 3.7 GHz). Each node executed its allocated simulations sequentially at maximum frequency. In total, this simulation infrastructure provided 36 cores dedicated to the experiment. ARDiS experiments ran directly on the target Intel i9-12900K platform, executing benchmarks sequentially on P-cores and E-cores across all VF configurations.

4.5.2 Results and Analysis. Table 5 presents the total wall-clock time required to complete the entire data collection process on each platform. The results reveal a significant difference in experimentation time.

ARDiS completed the data generation in 44 hours (1.8 days), compared to 362 hours (15.1 days) required by HotSniper running on a distributed system with 36 cores—an **8.2× speedup**. This represents a reduction of 318 hours, or approximately 13 days (88% decrease) in experimentation time for a single data collection experiment. Moreover, ARDiS achieves this speedup while using fewer than half the computational resources (16 cores vs. 36 cores) and running on the actual target

Table 5. Comparison of Training Data Generation Time between Simulation and Real Hardware

Platform	Compute	Total Cores	Time (hours)	Configurations	Speedup
HotSniper	1× Ryzen 9 3900X 3× Ryzen 7 2700X	36 cores	362 (15.1 days)	21	1× (baseline)
ARDiS	Intel i9-12900K (target platform)	16 cores	44 (1.8 days)	42 (P/E Cores)	**8.2**×

platform rather than requiring a separate simulation infrastructure. If normalized for computational resources, the effective advantage of ARDiS becomes even more apparent. In fact, ARDiS delivers approximately **18.5**× better time-to-results per core compared to the distributed simulation approach.

4.5.3 Implications for ML-based RM Research. As motivated in Section 1, this substantial speedup has critical implications for the practicality of ML-based resource management research workflows: ML model development typically requires multiple iterations of data collection as researchers refine sampling strategies, add features, or adjust experimental parameters. With HotSniper, each iteration would require over two weeks, even with distributed resources, making even modest refinements prohibitively time-consuming. On the other hand, ARDiS enables researchers to complete 8 iterations in the time HotSniper requires for one. Moreover, such time savings enable researchers to explore larger parameter spaces, additional benchmarks, or more fine-grained configurations that would be impractical with simulation. For instance, doubling the VF level granularity (42 levels instead of 21) would extend HotSniper experiments to 30 days but only 3.6 days with ARDiS. Furthermore, beyond time savings, ARDiS eliminates the need for dedicated simulation infrastructure. The HotSniper experiments required maintaining and coordinating four separate compute nodes, while ARDiS runs entirely on the target platform. This reduces both the hardware investment and the operational complexity of conducting RM research.

These results directly validate the core motivation of ARDiS presented in Section 1. High-fidelity simulators, despite their accuracy, impose computational costs that fundamentally limit the scope and pace of RM research. Even when distributed across multiple high-performance machines (36 cores total), simulation required over two weeks to complete experiments that ARDiS finished in under two days on a single platform. By eliminating the simulation overhead while maintaining measurement fidelity on real hardware, ARDiS transforms research workflows from impractical multi-week experiments into manageable multi-day ones.

Finally, this comparison represents a best-case scenario for simulation. The 362-hour simulation time assumes dedicated compute resources across four nodes with no system interruptions or resource contention. As RM techniques grow more sophisticated and require larger training datasets or more complex workload scenarios, the relative advantage of real hardware over simulation only increases. The scalability challenge is particularly acute for simulation: expanding the experimental scope requires proportionally more computational resources, whereas ARDiS scales naturally with the target platform's capabilities.

4.6 Comparison Against LUSH [36]

As will be discussed in Section 5, and as highlighted in Table 6, ARDiS offers a comprehensive and portable set of features and modules that *none of the state-of-the-art works have offered.* The closest work in the literature to our ARDiS, that is publicly-available and was maintained until recently is the LUSH framework [36]. Below, we present a functional and overhead comparison between our ARDiS and LUSH [36].

Table 6. Comparison of ARDiS with Existing RM Frameworks

Framework	Platform	Portable	DVFS	Mapping	Migration	Scheduling	Monitoring	Code Instr.	Open Src.
Custom Scripts [8, 16, 22, 24]	Real HW	✗	Varies	Varies	Varies	Varies	✓	Varies	N/A
ETFA'19 [18]	Real HW	✗	✗	✗	✗	✓	✓	Not Required	✗
LUSH [36]	Real HW	✗	✗	✓	✓	✓	✓	Not Required	✓†
POET [20]	Real HW	✓	✓	✓	✓	✗	✗	Required	✓†
Beeps [11]	Real HW	✗	✓	✗	✓	✗	✓	Required	✗
HotSniper [26]	Simulation	✓	✓	✓	✓	✓	✓	Not Required	✓
gem5 [5]	Simulation	✓	✓	✓	✓	✓	✓	Not Required	✓
ARDiS (This work)	**Real HW**	✓	✓	✓	✓	✓	✓	**Not Required**	✓

†No longer actively maintained or publicly unavailable.

4.6.1 Functional Comparison. Our ARDiS framework significantly outperforms LUSH across all critical dimensions of resource management frameworks. The most critical aspect of this superiority is the fact that LUSH [36] only supports single-threaded single-application execution. This feature limits the usability of the framework in any realistic modern workload execution scenario, where workloads of multi-threaded applications are executed in parallel, similar to executed using ARDiS in Figures 5 and 6. Moreover, the functionality richness and breadth of the LUSH framework are very limited. Compared to ARDiS's modern object-oriented architecture with modular components (Engine, DVFSPolicy, MappingPolicy, MigrationPolicy, Monitor, etc.), LUSH's is based on a monolithic C-based design with functionality limited to monitoring and task migration. In fact, LUSH offers the monitoring of a fixed set of hard-coded counters specific to the ODROID platform, in addition to only basic binary core selection between "little" and "big" cores, built specifically for the ARM big.LITTLE heterogeneous architecture on the same platform. On the other hand, through its generic configuration, ARDiS can be deployed on any host processor and provides granular monitoring modes at core, process, or thread-level, with extensive performance metrics collection.

This functional comparison highlights that ARDiS is a superior resource management framework, offering researchers and practitioners a comprehensive, extensible, powerful, and readily-portable framework that significantly advances the state-of-the-art in resource management.

4.6.2 Overhead Comparison. The authors of LUSH [36] report a runtime overhead of 0.625% when executing a simple microbenchmark single-application workload that alternates between CPU-intensive arithmetic operations and memory-bound array access patterns, running for 60 cycles total on a single core. During this experiment, the authors disabled all interventions of the *prediction* module that might trigger migrations, leaving the app to run on a *big* core from start to finish, while monitoring five PMCs. We have ported this microbenchmark to our ARDiS and run the same experiment where the application is pinned to a static core at a static VF level. We then compared it to an execution where the application is run on the same core at the same VF level outside of ARDiS, using *taskset* for core pinning and *time* to measure the execution time. The results of this experiment show that ARDiS introduces **a negligible overhead of only 0.18%**, compared to 0.625% reported by LUSH [36]. This represents **a 3.5× reduction in overhead** while proving superior resource management capabilities, as discussed in Section 4.6.1.

4.7 Runtime Overhead with Large Workloads

To evaluate the performance overhead incurred by monitoring PMCs with large workloads, we run 25 randomly selected workloads from the SPEC CPU 2006 benchmark suite at 100% core occupancy, both with and without the monitoring module enabled on our two evaluation platforms. We calculate the overhead as the average percentage difference between execution time without

monitoring (t_{OFF}) and with continuous monitoring enabled (t_{ON}):

$$Overhead = 100\% \times \frac{t_{ON} - t_{OFF}}{t_{OFF}}. \quad (1)$$

Our results indicate that ARDiS introduces a minimal overhead of approximately **1.61%** on the Jetson TX2 platform and **1.07%** on the Intel i9 setup, demonstrating negligible impact on overall performance.

5 Related Work

This section summarizes state-of-the-art RM techniques that use simulation and those that use real hardware. Additionally, we discuss the prior efforts to overcome limitations of real-hardware RM frameworks. Many approaches for resource management, especially in the multi-/many-core domain, employ hardware simulation due to its ease of use, flexibility, and reconfigurability [6, 15, 27, 28, 37]. In [28], the authors employ an ML technique leveraging DVFS for thermal management in a many-core architecture using the HotSniper Simulator [26]. The same simulation framework is used in [27], where a Neural Network model is used to predict an efficient task migration scenario. Using an interval-based simulator such as Sniper [7] helps researchers to quickly prototype the techniques at the cost of reduced accuracy compared to real hardware.

The other prominent simulator framework used in resource management research is gem5 [5, 23]. An example of a technique that incorporates this cycle-accurate framework is presented in [37] where DVFS is leveraged for energy modeling in an Edge Computing scenario. While gem5 is a cycle-accurate simulator, which means it is much closer to a real system, the cost of using such a tool is the simulation time, which can be impossibly large for exhaustively evaluating resource management techniques in complex workloads.

In order to deal with the shortcomings of the simulation frameworks, several approaches employ real hardware setups, based on custom scripts, to collect relevant execution metrics to then apply the resource management actions [8, 16, 22, 24]. In [22], the authors develop an ML-based technique aimed at improving power efficiency of CPUs. Thermal management is another common target for such techniques. In [8], the authors proposed an OS-level approach for this target in embedded/mobile ARM-based systems. The issue with most real-hardware techniques for resource management is that the results heavily depend on the underlying architecture. With custom setups and scripts, it becomes unfeasible to replicate the experiments performed by the authors. Moreover, lacking a unified framework for monitoring and controlling the system makes it hard to reconfigure and port the experiments to different execution environments.

To overcome the limitations of both simulation frameworks and custom real-hardware setups, a few works have tried to propose a unified execution framework. In [18], the authors introduce a framework that focuses on integrating ML techniques in schedulers in multi-core systems. However, their scope is mostly limited to scheduling, and the evaluation is rather limited to generic threads instead of real workloads such as those in benchmark applications. In [36], the authors present LUSH, a framework for user-level scheduling in heterogeneous multicore systems, designed to improve generality over [18]. The tool provides mechanisms for application monitoring using performance counters, prediction of system states, and thread migration strategies. However, the focus of this framework relies on the custom kernel-level monitoring functionality, with little to no support reported for custom resource management policies in addition to task migration (e.g., custom arrival times, application mapping, and DVFS). Earlier tools, such as POET [20] and Beeps [11], are based on API calls to aid in reporting the application's performance. Although useful from a monitoring perspective, these tools require instrumentation of the application's code, which makes them impractical for existing binary applications.

In contrast to previous approaches to resource management frameworks, ARDiS offers the repeatability, flexibility, portability, and reconfiguration of simulation tools with the realistic nature and real-world practicality provided by actual hardware platforms. Our framework offers a unified hub to design, launch, and evaluate resource management-based experiments, providing knobs completely at the user level to support modern resource management approaches.

6 Conclusion

We presented ARDiS, the first open-source, portable, and architecture-agnostic framework designed to unify the design, implementation, and evaluation of system-level resource management techniques directly on real hardware. Through extensive experimentation on two diverse hardware platforms, we demonstrated that ARDiS enables accurate, scalable, and reproducible evaluations with minimal monitoring overhead (less than 2%). By streamlining sophisticated RM tasks, including application characterization under varying power modes, ML-driven policy data generation, and replication of state-of-the-art approaches across multiple domains, ARDiS significantly reduces the complexity and effort traditionally associated with RM research, thereby empowering the design automation community to explore and innovate in system-level optimization with greater efficiency and reproducibility.

References

[1] 2021. Hybrid architecture (code name Alder Lake). *Intel*. Retrieved May 4, 2025 from https://www.intel.com/content/www/us/en/developer/articles/technical/hybrid-architecture.html

[2] 2017. Jetson TX2 Module. *NVIDIA*. Retrieved May 4, 2025 from https://developer.nvidia.com/embedded/jetson-tx2

[3] 2009. perf: Linux profiling with performance counters. Retrieved May 4, 2025 from https://perf.wiki.kernel.org/index.php/Main_Page

[4] Christian Bienia and Kai Li. 2009. Parsec 2.0: A new benchmark suite for chip-multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*. 37.

[5] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7. DOI: 10.1145/2024716.2024718

[6] Claudiu Buduleci, Arpad Gellert, Adrian Florea, and Remus Brad. 2024. Architectural and technological approaches for efficient energy management in multicore processors. *Computers* 13, 4 (2024), 84. DOI: 10.3390/computers13040084

[7] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. 2011. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulation. In *Proceedings of the High Performance Computing, Networking, Storage and Analysis*. ACM. DOI: 10.1145/2063384.2063454

[8] Nan Che, Weihua Chen, Puning Zhao, Fei Yu, Zhijun Li, Xing Gao, Yuandi Li, Xiaogang Cui, and Jie Cheng. 2024. OS-Level PMC-based runtime thermal control for ARM mobile CPUs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 7 (2024), 2023–2036. DOI: 10.1109/TCAD.2024.3360319

[9] Maxime Chéramy, Pierre-Emmanuel Hladik, and Anne-Marie Déplanche. 2014. Simso: A simulation tool to evaluate real-time multiprocessor scheduling algorithms. In *Proceedings of the 5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*. 6–p.

[10] Thomas Ebi, David Kramer, Wolfgang Karl, and Jörg Henkel. 2011. Economic learning for thermal-aware power budgeting in many-core architectures. In *Proceedings of the 7th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '11)*. Association for Computing Machinery, New York, NY, USA, 189–196. DOI: 10.1145/2039370.2039401

[11] Francisco Gaspar, Luis Taniça, Pedro Tomás, Aleksandar Ilic, and Leonel Sousa. 2015. A framework for application-guided task management on heterogeneous embedded systems. *ACM Transactions on Architecture and Code Optimization* 12, 4, Article 42 (2015), 25 pages. DOI: 10.1145/2835177

[12] Davy Genbrugge, Stijn Eyerman, and Lieven Eeckhout. 2010. Interval simulation: Raising the level of abstraction in architectural simulation. In *Proceedings of the 16th International Symposium on High-Performance Computer Architecture (HPCA)*. 1–12. DOI: 10.1109/HPCA.2010.5416636

[13] Jeferson González-Gómez, Mohammed Bakr Sikal, Heba Khdr, Lars Bauer, and Jörg Henkel. 2023. Smart detection of obfuscated thermal covert channel attacks in many-core processors. In *Proceedings of the 2023 60th ACM/IEEE Design Automation Conference (DAC)*. 1–6. DOI: 10.1109/DAC56929.2023.10247844

[14] Jeferson González-Gómez, Mohammed Bakr Sikal, Heba Khdr, Lars Bauer, and Jörg Henkel. 2024. Balancing security and efficiency: system-informed mitigation of power-based covert channels. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 11 (2024), 3395–3406. DOI: 10.1109/TCAD.2024.3438999

[15] Manjari Gupta, Lava Bhargava, and S. Indu. 2021. Dynamic workload-aware DVFS for multicore systems using machine learning. *Computing* 103, 8 (2021), 1747–1769.

[16] Ujjwal Gupta, Sumit K. Mandal, Manqing Mao, Chaitali Chakrabarti, and Umit Y. Ogras. 2019. A deep q-learning approach for dynamic management of heterogeneous processors. *IEEE Computer Architecture Letters* 18, 1 (2019), 14–17. DOI: 10.1109/LCA.2019.2892151

[17] John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

[18] Leonardo Passig Horstmann, José Luis Conradi Hoffmann, and Antônio Augusto Fröhlich. 2019. A framework to design and implement real-time multicore schedulers using machine learning. In *Proceedings of the 2019 24th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. 251–258. DOI: 10.1109/ETFA.2019.8869545

[19] Hengli Huang, Xiaohang Wang, Yingtao Jiang, Amit Kumar Singh, Mei Yang, and Letian Huang. 2022. Detection of and countermeasure against thermal covert channel in many-core systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 41, 2 (2022), 252–265.

[20] Connor Imes, David H. K. Kim, Martina Maggio, and Henry Hoffmann. 2015. POET: A portable approach to minimizing energy under soft real-time constraints. In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium*. 75–86. DOI: 10.1109/RTAS.2015.7108419

[21] Heba Khdr, Mohammed Bakr Sikal, Benedikt Dietrich, and Jörg Henkel. 2025. Towards the optimization of hardware efficiency through machine learning. In *Proceedings of the 2025 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*.

[22] Shivam Kundan and Iraklis Anagnostopoulos. 2020. A machine learning approach for improving power efficiency on clustered multi-processor system. In *Proceedings of the 2020 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–5. DOI: 10.1109/ISCAS45731.2020.9180474

[23] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, and Srikant Bharadwaj, et al. 2020. The gem5 Simulator: Version 20.0+. arXiv:2007.03152. Retrieved from https://arxiv.org/abs/2007.03152

[24] Sumit K. Mandal, Ganapati Bhat, Chetan Arvind Patil, Janardhan Rao Doppa, Partha Pratim Pande, and Umit Y. Ogras. 2019. Dynamic resource management of heterogeneous mobile platforms via imitation learning. *IEEE Transactions on Very Large Scale Integration Systems* 27, 12 (2019), 2842–2854. DOI: 10.1109/TVLSI.2019.2926106

[25] Theodoros Marinakis, Shivam Kundan, and Iraklis Anagnostopoulos. 2019. Meeting power constraints while mitigating contention on clustered multiprocessor system. *IEEE Embedded Systems Letters* 12, 3 (2019), 99–102.

[26] Anuj Pathania and Jörg Henkel. 2019. HotSniper: Sniper-based toolchain for many-core thermal simulations in open systems. *IEEE Embedded Systems Letters* 11, 2 (2019), 54–57. DOI: https://doi.org/10.1109/LES.2018.2866594

[27] Martin Rapp, Anuj Pathania, Tulika Mitra, and Jörg Henkel. 2021. Neural network-based performance prediction for task migration on S-NUCA many-cores. *IEEE Transactions on Computers* 70, 10 (2021), 1691–1704. DOI: 10.1109/TC.2020.3023022

[28] Martin Rapp, Mohammed Bakr Sikal, Heba Khdr, and Jörg Henkel. 2021. SmartBoost: Lightweight ML-driven boosting for thermally-constrained many-core processors. In *Proceedings of the 2021 58th ACM/IEEE Design Automation Conference (DAC)*. 265–270. DOI: 10.1109/DAC18074.2021.9586287

[29] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. Association for Computing Machinery, New York, NY, USA, 475–486. DOI: 10.1145/2485922.2485963

[30] Mohammed Bakr Sikal, Jeferson González-Gómez, Heba Khdr, and Jörg Henkel. 2025. Contention-aware forecasting of energy efficiency through sequence-based models in modern heterogeneous processors. In *Proceedings of the 2025 62nd ACM/IEEE Design Automation Conference (DAC)*.

[31] Mohammed Bakr Sikal, Heba Khdr, Martin Rapp, and Jörg Henkel. 2023. Machine learning-based thermally-safe cache contention mitigation in clustered manycores. In *Proceedings of the 2023 60th ACM/IEEE Design Automation Conference (DAC)*. 1–6. DOI: 10.1109/DAC56929.2023.10247708

[32] Mohammed Bakr Sikal, Heba Khdr, Lokesh Siddhu, and Jörg Henkel. 2024. ML-based thermal and cache contention alleviation on clustered manycores with 3-D HBM. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 11 (2024), 3614–3625. DOI: 10.1109/TCAD.2024.3438998

[33] Mohammed Bakr Sikal, Hassan Nassar, Heba Khdr, and Jörg Henkel. 2025. A dataset for LLM-based detection of power-wasters in routed FPGA netlists. In *Proceedings of the 2025 IEEE International Conference on LLM-Aided Design (ICLAD)*. 235–241. DOI: 10.1109/ICLAD65226.2025.00023

[34] Rafael Ubal, Byunghyun Jang, Perhaad Mistry, Dana Schaa, and David Kaeli. 2012. Multi2Sim: A simulation framework for CPU-GPU computing. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*. Association for Computing Machinery, New York, NY, USA, 335–344. DOI: 10.1145/2370816.2370865

[35] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*.

[36] Vasco Miguel Liang Xu, Liam White McShane, and Daniel Mossé. 2021. LUSH: Lightweight framework for user-level scheduling in heterogeneous multicores. In *Proceedings of the 2021 IEEE 14th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. 396–404. DOI: 10.1109/MCSoC51149.2021.00065

[37] Yahya H. Yassin, Magnus Jahre, Per Gunnar Kjeldsberg, Snorre Aunet, and Francky Catthoor. 2021. Fast and accurate edge computing energy modeling and DVFS implementation in GEM5 using system call emulation mode. *Journal of Signal Processing Systems* 93, 1 (2021), 33–48.

Received 26 May 2025; revised 5 October 2025; accepted 8 January 2026