Université de Mons
Faculte Polytechnique de Mons

Energy-optimal configurations for High-Performance Computing applications: automated low-impact characterization and performance optimization of shared-memory applications

Vitor Ramos Gomes da Silva

A dissertation submitted in fulfilment of the requirements of the degree of *Docteur en sciences de l'ingénieur et technologie*

Advisors
Prof. Carlos VALDERRAMA, UMons, Supervisor
Prof. Pierre MANNEBACK, UMons, Co-supervisor
Prof. Samuel XAVIER-DE-SOUZA, UFRN, Co-supervisor

Jury
Prof. Thierry DUTOIT, UMons, President
Prof. Sidi MAHMOUDI, UMons, Secretary
Prof. Demétrios COUTINHO, UFRN, Chair member
Eng. Lotfi GUEDRIA, CETIC, Chair member
Prof. Emanuel TRABES, UMONS, Chair member

August 24, 2024

Acknowledgements

I would like to thank my wife, Ingrid, my parents, Claudia and Paulo, and my sister, Tais, for their constant support and encouragement during my doctoral studies.

I would also like to thank Professor Carlos and Professor Samuel for their guidance, expertise, and encouragement. Their valuable comments and advice helped shape this research into its current form. I would also like to thank the entire research team at UMons and UFRN for the numerous collaborations that facilitated this research. Special thanks to my colleagues, Anderson Braulio and Alex Fortunato for their helpful suggestions and support during our numerous discussions.

I acknowledge the financial support provided by UMons, which made this research possible.

To all who contributed to this work, thank you.

Abstract

Energy consumption is key to enabling exascale High-performance Computing (HPC). However, energy-optimized hardware and software combinations could still be inefficient if the software operates poorly.

This work proposes a set of tools, models, and algorithms for energy optimization aimed at high-performance computing based on knowledge of the application and the specific hardware architecture. The main contributions of this work are.

A framework called Parallel Scalability Suite (PaScal Suite) automatically measures and compares multiple executions of a parallel application according to various scenarios characterized by input size, number of threads, cores, and frequencies. As a result, PascalSuite can automate designing application models with an overhead of less than 1%.

An entire system energy model based on the CPU frequency and the number of cores. The model aims to understand and optimize the energy behavior of parallel applications in HPC systems according to application parameters, such as the degree of parallelism, input load, and CPU parameters related to dynamic and static power.

A methodology that combines measurement data with a heuristic algorithm to provide insights into choosing the best phase divisions. Our heuristic can reduce the scan space from 10^{7000} to 10^2 with an average error of 10% and up to 38% reduction in energy consumption using optimal distribution compared to standard Linux DVFS.

A novel normalized time representation of the application characterizes the application parameters and model, named application fingerprint.

Contents

List of Figures

List of Tables

List of Acronyms

Various acronyms will be used throughout this dissertation to abbreviate frequent terms, some of which will even find usages across all sections. The expansion will be given at least on the first occurrence of each acronym in the text, but the following list of acronyms can be used as a reference if needed.

DPM Dynamic Power Management. iv, x, 3–5, 13, 52–54, 75–77, 85, 88, 91, 124, 140

DVFS Dynamic Voltage and Frequency Scaling. iv, x, 3–6, 13, 38, 52–54, 56, 75–77, 85, 88, 91, 92, 99, 124, 140

FINFET Fin Field-Effect Transistor. 59

HPC High Performance Computing. iv, viii, x, 2, 4, 5, 8, 11–13, 19–21, 23, 33, 38, 53, 54, 75, 76, 87, 91, 123–126, 140

IPMI Intelligent Platform Management Interface. viii, 14, 15, 17–19, 25, 29, 35, 67

MOSFET Metal-Oxide-Semiconductor Field-Effect Transistor. 59

MPE Mean Percentage Error. ix, x, 69, 72–74, 82

MSE Mean Squared Error. 71

MSR Model-Specific Registers. 19, 41

PMU Performance Monitoring Unit. 38, 39, 41, 43

RAPL Intel Running Average Power Limit. 14, 17, 19, 25, 29, 35

SVR Support Vector Regression. x, xii, 68, 70–74, 82

CHAPTER 1

Introduction

In this chapter, we present the motivations for this work, as well as its objectives and main contributions.

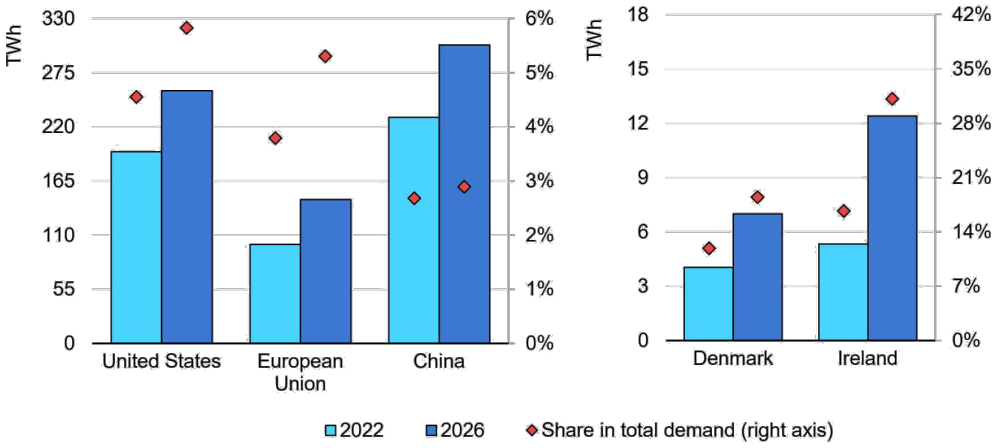
1.1. Motivation

In recent years, the importance of data center energy efficiency has significantly escalated due to its profound economic, environmental, and performance implications. The energy consumption of leading petaflop supercomputers, for instance, ranges from 1 to 30 MW TOP500 (2024) of electrical power, averaging around 2 MW, translating into substantial annual electricity expenses, often in the millions of dollars Ishfag (2012). In 2010, data center energy usage was estimated to account for between 1.1% and 1.5% of global electricity consumption Dayarathna (2016); Corcoran (2017); Iea (2021), resulting in environmental repercussions comparable to those of a nation like Argentina Mathew (2012). Moreover, in some scenarios, the expenditure on power surpasses the costs associated with hardware procurement Rivoire (2007). This trend is exacerbated by the doubling of energy expenses for operating a typical data center every five years Buyya (2013), making electricity bills a substantial financial burden for data center operators Poess (2008); Gao (2013). Consequently, data center energy efficiency has emerged as a primary concern for operators, often prioritized over traditional considerations such as availability and security.

According to the International Energy Agency, data centers in the United States, European Union, China, Denmark, and Ireland are expected to experience a considerable rise in electricity consumption from 2022 to 2026, with China and Ireland seeing the most substantial increases (see Figure 1.1). This surge in energy demand, coupled with the growing share of total electricity consumption attributed to data centers, underscores the urgent need for more efficient energy management strategies.

Moreover, there are various types of HPC architectures, each with unique challenges and opportunities for energy efficiency. In this work, we focus on cluster architectures, which have become the dominant HPC architecture over the last few decades, as illustrated in Fig. 1.2. The widespread adoption of cluster architectures is driven by several factors: they utilize well-known, widely-supported software, eliminate the need for specialized hardware, and benefit from the economies of scale, making them more cost-effective to maintain. This trend underscores the critical

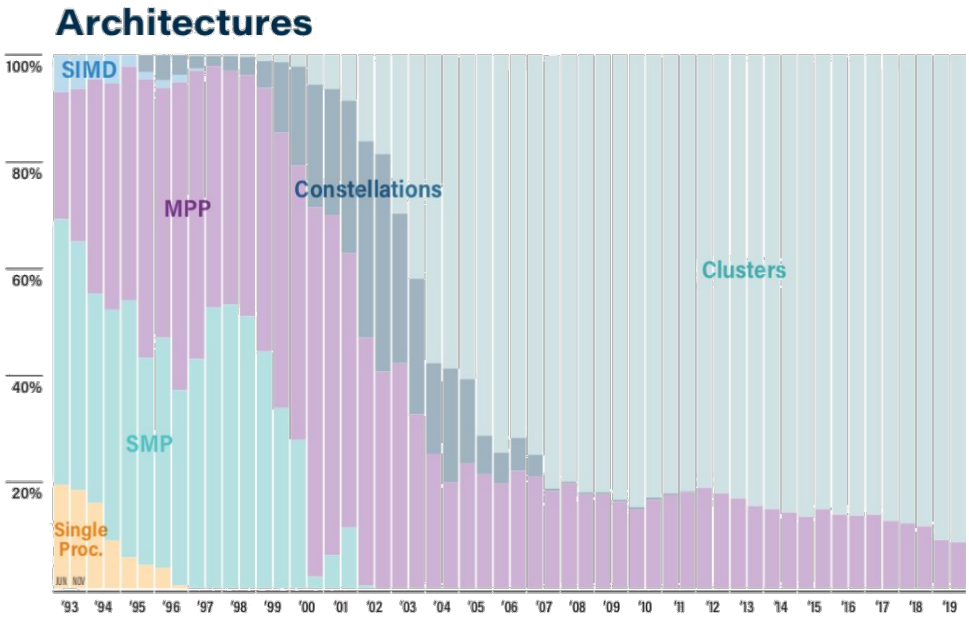
Figure 1.1: Estimated data center electricity consumption and its share in total electricity demand in 2022 and 2026, International Energy Agency.

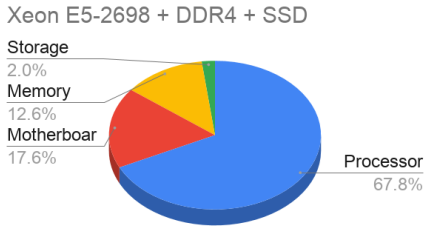


importance of optimizing energy efficiency within these shared memory systems.

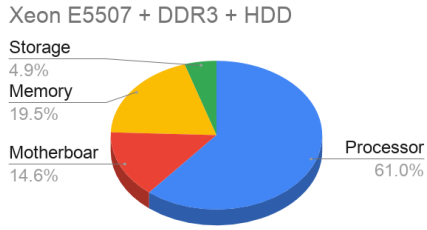
In the cluster HPC systems there are various strategies for achieving green computing have been devised, ranging from advancements in electrical materials to circuit design, systems integration, and software optimization. While these approaches may vary, they share a common objective: to significantly reduce overall system energy consumption without compromising performance. Notably, the processor and main memory are the primary contributors to power consumption, collectively accounting for a significant portion, as shown in Fig. 1.3. The processor alone can consume up to 50% of the total energy Fan (2007); Barroso (2007); Malladi (2012). Consequently, modern processors are equipped with numerous power management features Rotem (2012); Brown (2005); Hackenberg (2015); Intel (2020); Cardoso (2017), including Dynamic Power Management and Dynamic Voltage and Frequency Scaling. DPM encompasses a suite of techniques aimed at achieving energy-efficient computing by deactivating or reducing the performance of system components when they are idle or underutilized Shuja (2012); Benini (2000). On the other hand, DVFS enables the adjustment of frequency and voltage in real time based on current workload demands.

Figure 1.2: HPC Architectures





(a) Our system, built in 2016 and equipped with two Intel Xeon E5-2698, 128 GB of DDR4 memory, and SSD as storage.



(b) Study case Malladi (2012), built in 2012 and equipped with two Xeon E5507, 32GB of DDR3 memory and HDD as storage.

Figure 1.3: Power breakdown of a typical node of an HPC cluster at full use. It is noteworthy that the primary components contributing to energy consumption are processor execution and memory usage.

DVFS operates on the principle that frequency and power exhibit a near-cubic relationship Dayarathna (2016); Ishfag (2012). This relationship suggests that lowering the CPU frequency results in a linear decrease in performance and a nearly cubic reduction in power consumption, potentially leading to a near-square reduction in CPU energy usage. Consequently, significant energy savings can be realized through frequency control alone, contingent upon the system and its architecture. However, while promising, determining the optimal voltage and frequency settings for running applications remains a challenge for system software. Failure to do so not only risks performance degradation but could also increase energy consumption Ishfag (2012).

Reducing the CPU frequency extends the execution time, subsequently increasing the energy consumption of other system components like memory and disks. Moreover, there's an associated overhead of time and energy with voltage and frequency switching, further complicating the selection process. Thus, identifying the most suitable voltage and frequency settings for all scenarios presents a non-trivial task. Consequently, since its inception in 1994, Ishfag (2012), extensive research has been dedicated to developing DVFS algorithms.

The DPM technique can yield substantial energy savings, particularly in systems characterized by high static power or prolonged periods of inactivity. In such scenarios, the challenge lies in determining the optimal timing and selection of components for activation or deactivation. Reports indicate that DPM has achieved energy savings of up to 70% Shuja (2012); Benini (2000).

While these power-saving techniques hold the potential to reduce overall system energy consumption, they may introduce performance compromises, necessitating a careful balance to develop more energy-efficient algorithms. Consequently, this study delves into whether constructing an energy consumption model for an application can result in significant energy savings. By understanding the intricacies of how an application interacts with system resources and consuming components, it becomes feasible to optimize energy utilization without sacrificing performance. Through this investigation, we aim to uncover insights that can inform the development of strategies to achieve the delicate equilibrium between energy efficiency and computational performance.

We propose an analytical energy model tailored to a given applica-

tion, contingent upon two primary control variables prevalent in most HPC systems: CPU operating frequency and the number of active cores. This model comprises three application-specific parameters and three architecture-related parameters. The application parameters encapsulate aspects such as parallelism percentage and input size, while the system architecture parameters encompass dynamic, static, and leakage power characteristics, inherently tied to power and technology considerations. With this customized approach, our objective is to establish a comprehensive framework for understanding and optimizing energy consumption, specifically tailored to the distinctive features of individual applications and system architectures.

1.2. Objectives

The overarching goal of this thesis is to advance solutions harnessing the full capabilities of involved systems, leveraging insights into architectures and applications. Specifically, the objectives are outlined as follows:

- Develop a framework facilitating and automating the modeling process for applications and architectures while maintaining transparency and non-interference with the application's operation.
- Propose an analytical energy model tailored to HPC systems' applications. This model should optimize energy consumption within these systems while enabling in-depth analysis through analytical equations.
- Introduce a heuristic algorithm designed to adapt to fluctuations in application behavior and system dynamics. This algorithm will consider evolving conditions to enhance energy efficiency in HPC environments.

1.3. Contributions

In this thesis, the contributions are delineated as follows.

In Chapter 3, we introduce a novel tool designed for the automated

measurement and comparison of multiple executions of parallel applications across diverse scenarios. These scenarios encompass variations in input arrangements, thread counts, core allocations, and frequencies. Unlike existing performance analysis tools, our proposed tool addresses critical gaps by providing specialized features necessary for comprehensively understanding scalability trends across computational resources. Importantly, it achieves this with minimal intrusion, imposing less than 1% overhead.

Chapter 4 presents a comprehensive full-system energy model predicated on CPU frequency and core count. The model is crafted to elucidate and optimize the energy dynamics of parallel applications within HPC systems. It incorporates application-specific parameters, such as parallelism degree, and CPU characteristics of dynamic and static power. Unlike conventional models, ours integrates frequency and core count within the same equation to estimate energy consumption for a given application configuration. This model serves as a foundational framework for addressing optimization challenges related to DVFS and DPM. Furthermore, by considering both frequency and active cores, it facilitates the analysis of each parameter's contribution to overall energy consumption. In essence, our model offers a robust foundation for refining energy optimization strategies in HPC environments, thereby enhancing the efficiency and sustainability of computational systems.

Chapter 5 introduces a novel methodology that merges empirical measurement data with a heuristic approach to guide the selection of optimal phase divisions. Our heuristic effectively reduces the scan space from an astronomical 10^{7000} to a manageable 10^2, with an average error rate of 10%. Moreover, it achieves up to a 38% reduction in energy consumption compared to standard Linux DVFS by identifying the optimal distribution of phases. Furthermore, we evaluate the trade-offs associated with excessive phase divisions and the ensuing overhead. Notably, our analysis reveals practical constraints on the number of phases that an application can effectively leverage, establishing a lower limit on the minimum number of phases required for optimal performance.

1.4. Organization

The document is organized as follows:

In Chapter 2, this chapter lays down fundamental concepts essential for the thesis, including definitions, system configurations, and benchmarks used throughout the study. It provides a crucial foundation for understanding the subsequent chapters.

Next Chapter 3 provides a comprehensive exposition of the PaScal Analyzer as a framework for automating energy analysis and experiments. This chapter details its functionalities and associated validation process, affirming its efficacy in practical applications.

Subsequently, Chapter 4 introduces the proposed energy model. Here, the intricacies of the model are elucidated, along with its formulation and validation methodologies. This chapter is pivotal for grasping the analytical framework employed in the study.

Among these chapters, the PaScal Analyzer Framework (Chapter 3) and the proposed energy model (Chapter 4) presents practical applications and potential to significantly impact energy analysis and optimization in computational systems.

Following that, Chapter 5, Optimization of Phase Divisions, presents a heuristic algorithm tailored to optimize phase divisions within applications. This chapter outlines the algorithm's design and its integration with empirical data to achieve energy-efficient outcomes.

Finally, Chapter 6 encapsulates the conclusions drawn from this thesis. It synthesizes the key findings, contributions, and implications of the research conducted, offering insights for future exploration in the field.

CHAPTER 2

Theoretical background

This chapter serve to elucidate key concepts and configurations essential for comprehending the subsequent discussions and analyses in this thesis. By providing clarity on fundamental terms, system configurations, and performance metrics, we aim to establish a solid foundation for exploring the intricacies of data center energy efficiency and performance optimization.

The chapter further elucidates the architectural details of HPC systems, encompassing components such as processors, memory subsystems, and interconnects. It also discuss varius aspects of applications that run on these systems.

Furthermore we devle into the intricacies of CPU performance states, elucidating how dynamic frequency scaling operates to regulate power consumption the dirverse sources available for monitoring power consumption, and other metrics we can use ranging from hardware sensors to software-based monitoring tools, we gain insight into the methodologies employed to quantify energy usage across various system components.

Lastly, we explore the selection criteria for applications utilized in experimental analyses throughout this thesis. By examining the characteristics and complexities of these applications, we lay the groundwork for conducting empirical investigations into performance optimization and energy efficiency in high-performance computing environments.

2.1. High-Performance Computing Architectures

High Performance Computing (HPC) (HPC) systems are designed to solve complex computational problems by leveraging the collective power of multiple processors, memory modules, and storage systems. These systems are typically composed of interconnected nodes, each containing one or more CPUs, memory, and sometimes local storage. The architecture of HPC systems can vary widely, but they generally fall into one of the following categories:

- **Symmetric Multiprocessing (SMP)**: In an SMP system, all processors share a single memory space and communicate through a common bus. This architecture is simple but can become a bottleneck as the number of processors increases due to contention for shared resources.
- **Massively Parallel Processing (MPP)**: MPP systems consist of multiple processors, each with its own private memory, connected through a high-speed interconnect. This allows for greater scalability, as each processor can operate independently, but requires explicit message passing (e.g., using MPI) to coordinate between processors.
- **Distributed Memory Systems (Clusters)**: Similar to MPP, distributed memory systems consist of nodes with private memory, but the nodes are connected over a network, often with lower bandwidth and higher latency than in MPP systems. This architecture is common in large-scale clusters.
- **Non-Uniform Memory Access (NUMA)**: NUMA is a hybrid architecture where memory is distributed among processors, but processors can access memory located on other processors. Memory access time depends on the memory's location relative to the processor, with local memory being faster to access than remote memory.

These architectural differences have significant implications for performance and energy consumption. Optimizing energy efficiency in HPC

systems requires a deep understanding of the underlying architecture, particularly how computational workloads are distributed and how memory access patterns affect performance.

As already mentioned in the previous section Chapter 1, this work focous on clusters, the most commonly used architecture.

2.2. Parallelism in HPC

Parallelism is a fundamental characteristic of HPC systems, enabling them to perform large-scale computations by dividing tasks across multiple processors. There are several forms of parallelism commonly exploited in HPC:

- **Data Parallelism**: This form of parallelism involves distributing different chunks of data across multiple processors, with each processor performing the same operation on its chunk. Data parallelism is particularly effective in applications where the same operation needs to be applied repeatedly to large datasets.
- **Task Parallelism**: In task parallelism, different processors perform different tasks simultaneously. This approach is well-suited to workflows where tasks can be executed independently or where tasks have varying computational requirements.
- **Pipeline Parallelism**: Pipeline parallelism involves dividing a task into stages, with each stage being processed by a different processor or group of processors. This is common in applications like video processing, where data flows through multiple stages of transformation.
- **Hybrid Parallelism**: Many HPC applications use a combination of data, task, and pipeline parallelism to optimize performance. For example, an application might use data parallelism within each node and task parallelism across nodes in a distributed memory system.

Managing parallelism effectively is crucial for optimizing both performance and energy efficiency. In systems with NUMA architecture, for

example, thread placement and memory allocation strategies must be carefully managed to minimize costly remote memory accesses, which can degrade performance and increase power consumption.

We are going to take advantage of this information to build specialized models that considers the characteristics of the applications that runs on those systems, this will be detailed in Section 4.5.

2.3. Energy Optimization in HPC Systems

Energy consumption is a critical concern in HPC due to the significant power requirements of large-scale computing systems. Several strategies have been developed to optimize energy usage without sacrificing performance:

- **Dynamic Voltage and Frequency Scaling (DVFS)**: DVFS is a widely used technique in HPC systems where the voltage and frequency of a processor are adjusted dynamically based on workload demands. Lowering the frequency reduces power consumption but also decreases performance, so DVFS must be carefully managed to balance energy savings with computational throughput.

- **Dynamic Power Management (DPM)**: DPM involves selectively powering down or reducing the power usage of certain components (e.g., CPU cores, memory modules, or entire nodes) during periods of low activity. This can lead to significant energy savings, particularly in workloads with irregular computational demands.

- **Power-Aware Scheduling**: In power-aware scheduling, tasks are assigned to processors in a way that minimizes energy consumption while meeting performance targets. This might involve scheduling tasks on processors that are already active, thereby avoiding the energy cost of waking up idle processors, or prioritizing energy-efficient processors for less demanding tasks.

- **Thermal Management**: High energy consumption in HPC systems often leads to increased heat generation, which can impact both performance and reliability. Thermal management techniques,

such as adaptive cooling and thermal-aware task scheduling, are used to control the temperature of the system while optimizing energy usage.

The energy optimization strategies discussed above are integral to your research. By leveraging a combination of DVFS, DPM, and power-aware scheduling, this work seeks to reduce the energy consumption of HPC systems while maintaining or even improving computational performance. The applications of those techniques will be detailed in Section 4.12.

2.4. Case-Study Architecture

The experiments conducted in this thesis were executed on a single computer node featuring two Intel Xeon E5-2698 v3 processors, each equipped with sixteen cores and two hardware threads per core. The architecture of the system is illustrated in Fig. 2.1. The processors operated at a maximum non-turbo frequency of 2.3 GHz, and the node boasted a total physical memory capacity of 128 GB (8×16 GB). Throughout the experiments, turbo frequency and hardware multi-threading functionalities were disabled to maintain consistency. The operating system employed was Linux CentOS 6.5, running kernel version 4.16.

The Linux kernel offers various power management policies depending on the driver in use (see Section 2.4.1). In our setup, utilizing the default acpi-cpufreq driver, the available governor options included are Powersave, Performance, Ondemand, Conservative, and Userspace. Each governor implements a distinct policy for frequency selection. In our investigations, the frequency control was achieved using the Userspace governor, enabling the application to set the CPU frequency. Additionally, core control was managed by modifying relevant system files with the default CPU-hotplug driver.

Furthermore, the architecture was equipped with the Intelligent Platform Management Interface (IPMI) and the Intel Running Average Power Limit (RAPL) Section 2.4.2, which comprises a set of interfaces facilitating out-of-band management of computer systems and monitoring of platform status via the local network Schwenkler (2006). IPMI enables the monitoring of various system variables and resources, such as tem-

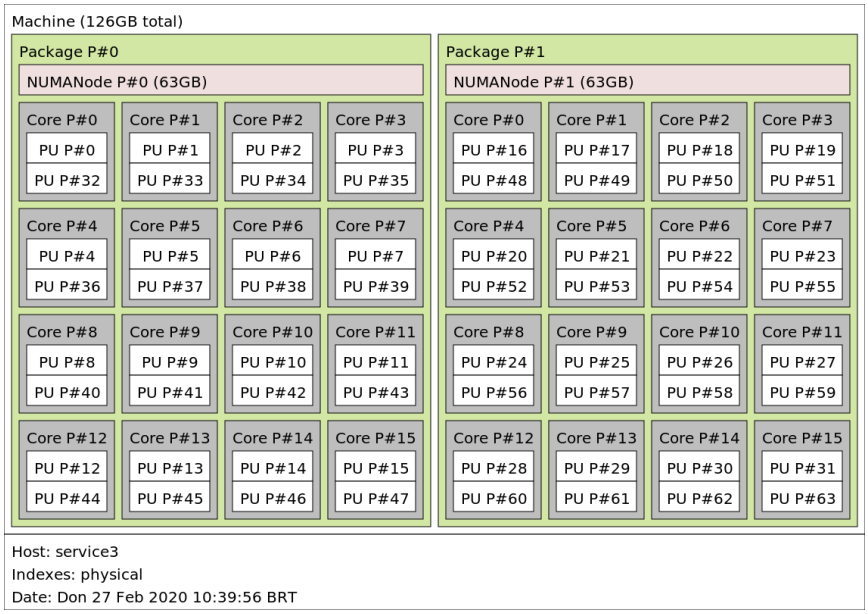


Figure 2.1: Node architecture: 2 Intel Xeon E5-2698 v3 processors, 16 cores each, 2 hardware threads per core, total physical memory capacity of 128 GB (8×16 GB) (the image was made with the lstop application).

perature, voltage, fan speed, and power supplies, leveraging independent sensors integrated into the hardware. This capability enhances system monitoring and management, contributing to overall operational efficiency and reliability.

2.4.1 Frequency Control

In contemporary computing systems, the regulation of processor frequency can be achieved through a combination of hardware and software mechanisms. A pivotal standard in this realm is the Advanced Configuration and Power Interface (ACPI), embraced by operating systems to configure power management settings for hardware components.

Within the ACPI framework, two key states, crucial for Dynamic Frequency Scaling are defined: "C" and "P". These states optimize energy consumption by modulating the processor activity based on workload demands.

The "P" state, activated during processor operation, encompasses multiple levels denoted as P_0 through P_n. At each level, both frequency and voltage are adjusted, with P_0 representing the highest possible frequency and voltage, gradually decreasing through subsequent levels until reaching P_n, where both are minimized. The transition between P states is contingent upon processor utilization, with the processor transitioning to a higher or lower state based on the workload intensity. The specific thresholds for state transitions vary by manufacturer.

Conversely, during idle periods, the processor enters "C" states, beginning with C_0, where it remains fully active, and progressing through C_1 to C_n as activity diminishes. At each level, various features are disabled to conserve power. While the ACPI standard delineates functionalities disabled between levels C_1 and C_3, other levels are manufacturer-specific.

Table 2.1 illustrates the functionalities disabled across C states as per the ACPI standard, providing insight into the power-saving mechanisms employed during periods of reduced processor activity.

In the C states, the higher the level the greater the energy savings, but returning to the fully functional level is more difficult. Fig. 2.2 best illustrates these changes of states, in which we can see which parts of the circuits are deactivated and the latency and energy required to return to the active state (Wakeup Time and Transition Energy).

Mode	Name	Functionality
C0	operating state	Active processor
C1	Halt	Stop executing instructions
C2	Stop-Clock	Disable the internal clock
C3	Sleep	Disable cache coherence

Table 2.1: Advanced Configuration and Power Interface (ACPI) C states

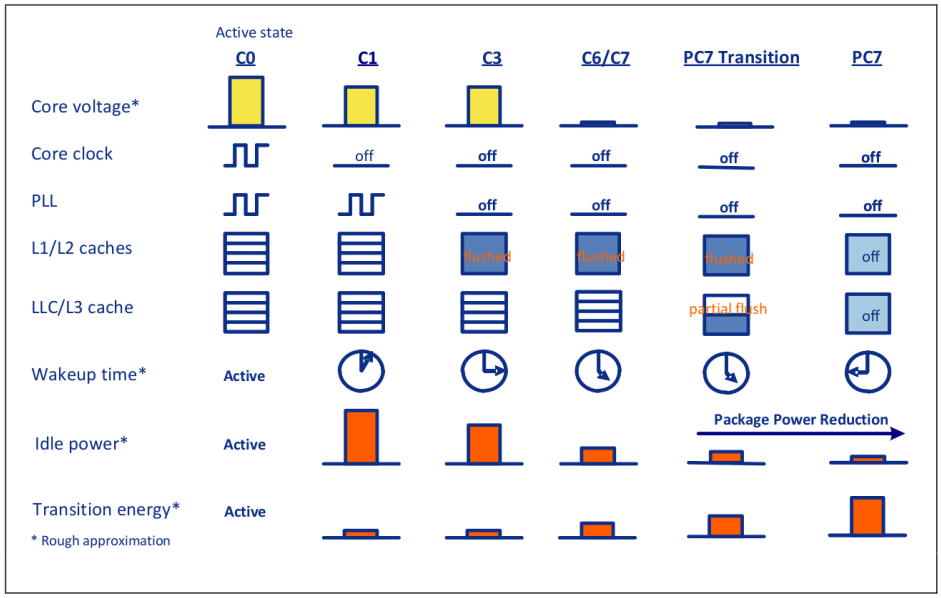


Figure 2.2: Illustration of C states: managed components and states' transition cost; source: `https://www.thomas-krenn.com/en/wiki/Processor_P-states_and_C-states`

In Linux, the management of the CPU frequency is facilitated through various tools and modules, providing users with granular control over performance and power consumption. One of the primary modules for this purpose is the *acpi-cpufreq*, which offers direct manipulation of CPU frequency settings through system files.

Acpi-cpufreq leverages a set of policies to dynamically adjust CPU frequency based on workload demands. These policies are accessible to users via the Linux filesystem, typically located under the /sys/devices/system/cpu/cpu*/cpufreq/ directory. Within this directory, users can find files corresponding to different frequency scaling governors, such as scaling_governor, which determines the current policy in use.

The available scaling governors include:

- Performance: This governor sets the CPU frequency to the maximum allowable value, ensuring optimal performance at all times. It's suitable for tasks that require maximum processing power, albeit at the expense of increased power consumption.

- Powersave: In contrast, the Powersave governor aims to minimize power consumption by setting the CPU frequency to the lowest possible value. This mode is ideal for scenarios where power efficiency takes precedence over performance.

- Userspace: The userspace governor grants users direct control over CPU frequency settings. By writing frequency values to the scaling_setspeed file, users can manually adjust CPU frequency to suit specific workload requirements.

- Ondemand: The Ondemand governor adjusts the CPU frequency dynamically based on processor load. As the workload increases, the governor scales up CPU frequency to meet demand, ensuring optimal performance while conserving power during idle periods.

- Conservative: Similar to the Ondemand governor, the conservative governor adjusts CPU frequency based on workload. However, it does so more gradually, avoiding sudden frequency changes to minimize power fluctuations.

Users can select the desired governor by writing its name in the scaling_governor file. Additionally, they can adjust other parameters, such as frequency scaling thresholds and transition latencies, to fine-tune performance and power efficiency according to specific use cases.

This comprehensive suite of frequency scaling governors empowers Linux users to effectively manage CPU performance and power consumption, enabling optimization for a wide range of computing scenarios.

2.4.2 Power consumption monitoring

In this work, we monitor power consumption by leveraging two key interfaces: the (Intel Running Average Power Limit) and the Intelligent Platform Management Interface, as detailed in IPMI (2013).

IPMI

IPMI, as outlined in IPMI (2013), encompasses a suite of specifications designed for autonomous subsystems, providing management and monitoring capabilities independent of processors, firmware, and operating systems. This enables system administrators to manage servers remotely. This feature is particularly beneficial given that server environments are often located in remote or inhospitable locations characterized by low temperatures and high levels of noise from ventilation systems. Remote management via IPMI allows administrators to perform tasks such as powering servers on and off, accessing the BIOS remotely, and reinstalling systems in case of critical failures.

As illustrated in Fig. 2.3, IPMI consists of critical components, including sensors that monitor various system parameters such as temperature, voltage, fan speed, and power supply. Communication with the IPMI network is facilitated through either the HTTP protocol or specialized tools like *ipmitool* provided by manufacturers. *ipmitool* grants network access to IPMI functionality, allowing administrators to monitor platform health in real time using sensor data. This capability provides valuable insights into system health and performance, facilitating real-time management of the server infrastructure without requiring constant manual oversight.

Our framework communicates seamlessly with IPMI via HTTP through the Baseboard Management Controller (BMC) network, enabling auto-

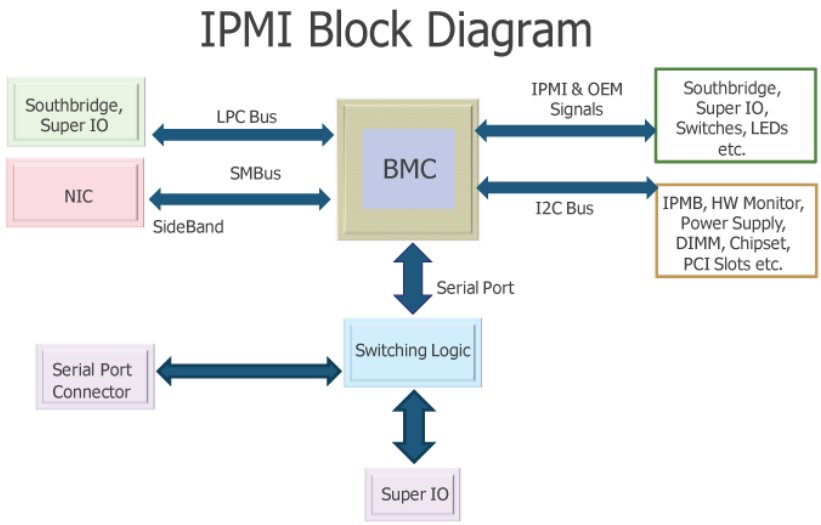


Figure 2.3: Diagram illustrating the key components and their interconnections within the IPMI architecture. [source `https://pt.wikipedia.org/wiki/Intelligent_Platform_Management_Interface`]

mated retrieval of power and sensor data. Leveraging this functionality, our tool streamlines system metric monitoring, covering power consumption and various sensor readings, reducing the need for constant manual intervention. This automated data collection enhances efficiency and facilitates real-time analysis of server health and performance, empowering administrators to optimize system operation and resource utilization through informed decisions.

RAPL

Modern Intel microprocessors, based on the SandyBridge architecture, include the RAPL interface Rotem (2012); Hahnel (2012); Hackenberg (2015) aimed at regulating chip energy usage while maintaining peak performance. This interface provides energy measurement capabilities through an integrated circuit that estimates power consumption based on a model driven by architectural event counters for all components.

The energy estimates proposed by RAPL have been validated by Intel. In addition, RAPL provides temperature readings and current leak models. These estimates are accessible through model-specific registers (MSR) and are updated every few milliseconds. The processor's MSR registers allow direct access to performance counter values, and power limits can be specified within defined time windows. The operating system offers a simplified interface for managing data, especially in cases of overflows. Beyond energy information, extensive performance parameter data is available, including cache hit rates, processor frequency, and branch instructions.

Our framework incorporates the capability to retrieve data from the RAPL interface, enhancing its effectiveness in monitoring and managing energy consumption in Intel-based microprocessors.

2.4.3 Performance Counters

Performance counters, also known as hardware counters or performance monitoring counters (PMCs), are specialized registers built into modern processors. These counters track a variety of low-level events and metrics related to processor and memory subsystem activities, providing detailed insights into system performance. By analyzing data from these

counters, developers and researchers can optimize software and hardware configurations for improved performance and energy efficiency in HPC environments.

Types of Performance Counters

Performance counters can be broadly categorized based on the type of events they monitor:

- **CPU Performance Counters**: These counters track events related to CPU operations, such as the number of instructions executed, cache hits and misses, branch predictions, pipeline stalls, and cycles spent in different execution states. For example, an important counter is the `Instructions Retired` counter, which counts the number of instructions that have been completed successfully.
- **Memory Performance Counters**: These counters monitor memory-related events, such as memory accesses, cache coherence events, memory bandwidth utilization, and NUMA-related metrics. Metrics like `Last Level Cache (LLC) Misses` are critical for understanding the efficiency of memory usage in HPC applications.
- **Power and Thermal Counters**: Some modern processors include counters that estimate power consumption and thermal behavior. For example, Intel's Running Average Power Limit (RAPL) counters provide energy consumption estimates for different parts of the processor, such as the CPU package, cores, and DRAM.
- **I/O Performance Counters**: These counters track events related to input/output operations, including disk accesses, network transfers, and peripheral device usage. These metrics are especially important in data-intensive HPC applications where I/O can become a bottleneck.
- **GPU Performance Counters**: For systems equipped with GPUs, performance counters monitor metrics related to GPU utilization, memory usage, and execution efficiency. These are particularly relevant in heterogeneous HPC systems where both CPUs and GPUs are used for parallel processing.

Utilizing Performance Counters in HPC

In the context of HPC, performance counters serve multiple purposes:

- **Performance Tuning**: By analyzing the data collected from performance counters, developers can identify performance bottlenecks and inefficiencies in their code. For example, high cache miss rates might indicate that data is not being accessed in a cache-friendly manner, leading to performance degradation. Optimizing such issues can significantly enhance computational throughput.
- **Energy Efficiency**: Performance counters provide essential data for energy optimization strategies. By correlating performance metrics with energy consumption (e.g., using RAPL counters), researchers can identify trade-offs between performance and power usage, enabling more informed decisions about frequency scaling and power management.
- **Application Profiling**: Performance counters allow for detailed profiling of applications, enabling developers to understand the behavior of their software on a granular level. This is particularly useful in identifying phases of computation that are either CPU-bound, memory-bound, or I/O-bound, which can then be optimized independently.
- **System Monitoring and Debugging**: Performance counters are also useful for real-time system monitoring and debugging. They help detect anomalies such as unexpected stalls, excessive branching, or inefficient resource utilization, which can be indicative of underlying hardware or software issues.

2.5. Case-Study Applications

To showcase the effectiveness of our power models and optimizations, we selected applications from three different benchmark suites to form the basis of our case study. These benchmarks were chosen for their suitability for High Performance Computing environments and their coverage of diverse use cases within such systems. Additionally, they are

widely studied in academia and used in industry. The chosen applications are Black-Scholes, Bodytrack, Canneal, Dedup, Fluidanimate, Freqmine, Raytrace, Swaptions, Vips and x264 from the PARSEC parallel benchmark suite version 3.0 Bienia (2008) (`https://parsec.cs.princeton.edu/download.htm`; accessed on 20 February 2020), OpenMC Romano (2015) and LINPACK (HPL) Dongarra (1988).

The PARSEC benchmark suite is specifically tailored to address emerging workloads, aiming to represent the next generation of shared-memory programs for chip multiprocessors. Spanning a diverse array of domains, including financial analysis, computer vision, engineering, enterprise storage, animation, similarity search, data mining, machine learning, and media processing, PARSEC offers a comprehensive coverage of modern computing applications. In contrast, the Linpack Benchmark assesses a computer's floating-point rate of execution by solving a dense system of linear equations. Widely adopted in the industry, the Linpack Benchmark serves as a standard measure for ranking supercomputers based on their computational performance. OpenMC, on the other hand, is a community-developed Monte Carlo neutron and photon transport simulation code. Renowned for its versatility, OpenMC is capable of performing fixed source, k-eigenvalue, and subcritical multiplication calculations on models constructed using either constructive solid geometry or CAD representation. Its adoption of a hybrid MPI and OpenMP programming model aligns well with the conventions of high-performance computing environments, making it an ideal candidate for evaluating system performance and efficiency in such contexts.

Program	**Domain**	**Parallelization**	**Granularity**	**Data Sharing**	**Data Exchange**
blackscholes	Financial Analysis	data-parallel	coarse	low	low
bodytrack	Computer Vision	data-parallel	medium	high	medium
canneal	Engineering	unstructured	fine	high	high
dedup	Enterprise Storage	pipeline	medium	high	high
facesim	Animation	data-parallel	coarse	low	medium
ferret	Similarity Search	pipeline	medium	high	high
fluidanimate	Animation	data-parallel	fine	low	medium
freqmine	Data Mining	data-parallel	medium	high	medium
raytrace	Rendering	data-parallel	medium	low	medium
streamcluster	Data Mining	data-parallel	medium	high	low
swaptions	Financial Analysis	data-parallel	coarse	low	medium
vips	Media Processing	data-parallel	coarse	low	medium
x264	Media Processing	pipeline	coarse	high	high
OpenMC	Simulation	hybrid	medium	high	high
LINPACK	Numerical Analysis	data-parallel	coarse	medium	low

CHAPTER 3

A Framework For Automated Energy Analysis and Experiments

Crafting accurate energy models demands an in-depth analysis across various execution scenarios. This challenge requires a profound understanding of parallel systems and the analytical tools that dissect their performance as well as energy consumption profiles. Despite the abundance of tools available, a gap remains in their ability to provide a balanced focus on both performance metrics and energy consumption insights, essential for holistic optimization strategies. To bridge this gap, we introduce a novel measurement framework specifically designed to analyze energy efficiency and parallel scalability.

Our proposed framework, detailed in our publications Silva (2019, 2022) focuses on a holistic view of software performance and energy efficiency, incorporating tracing, profiling with performance counters, and post-mortem analysis. The framework automates the analysis of parallel applications across a broad range of configurations by automatically identifying parallel regions in binaries, calculating energy consumption for each, and aggregating data to reduce memory and disk usage, all while maintaining high accuracy and low overhead of less than 1%.

3.1. Pascal Analyzer: An Advanced Framework for Energy Efficiency Analysis and Scalability

Given the emphasis on energy efficiency within the field of parallel computing, optimizing parallel programs for high-performance systems necessitates a deep understanding of their energy consumption patterns. Performance measurement and analysis tools play a crucial role in this optimization process by enabling developers to discern execution characteristics, identify energy inefficiencies, and isolate the computational behaviors that significantly impact the program's energy consumption. The primary goal of these tools is to guide developers in implementing enhancements that not only boost program efficiency but also reduce energy usage.

The complexity of modern parallel systems makes the task of accurately identifying and mitigating energy consumption bottlenecks particularly challenging Huck (2007); Islam (2019); Weber (2019). Developers must be adept at comparing various measurements across different execution configurations to pinpoint these bottlenecks. This process is often arduous and demands a comprehensive grasp of the domain, parallel systems, and the intricacies of performance and energy measurement tools Bergel (2019); Weber (2019); Malony (2005); Geimer (2010); Malony (2006); Adhianto (2010); Miller (1995); Galobardes (2015); Pillet (2007); Islam (2019). These tools, each with unique measurement strategies, metrics, and focus areas, require mastery to effectively harness their capabilities Brink (2020); Huck (2007).

Although many existing tools provide valuable insights into program behavior through a detailed set of metrics, their utility in analyzing energy efficiency and parallel scalability specifically in the context of energy consumption can be limited Bergel (2019); Silva (2018). Traditional tools tend to focus on analyzing single execution runs, whereas a more holistic understanding of energy efficiency often necessitates examining the program across multiple configurations of the target execution platform. From an energy perspective, scalability analysis benefits more from comparing key data points across varied configurations rather than collecting

extensive, fine-grained data from single configurations.

The intrinsic overhead associated with traditional analysis tools can skew the observed energy consumption data. Some studies indicate that this overhead can constitute a significant portion of the program's runtime energy consumption Eriksson (2016), reaching 40%, thus obscuring the true energy efficiency landscape. An analytical approach that prioritizes the collection of only essential data can mitigate this issue, reducing the degree of intrusion and simplifying the tool's usage for developers.

In this light, the contributions made through this thesis, particularly with the introduction of PaScal Analyzer, are pivotal. This novel framework contributes the approach to energy efficiency analysis in parallel computing by automating the measurement and comparison of parallel application executions across various scenarios with an exceptionally low overhead. By focusing on energy models and offering specialized features for an in-depth exploration of energy consumption patterns across different computational scenarios, PaScal Analyzer directly addresses the challenges of existing performance analysis tools Roberts (2017); Eastep (2017); Hackenberg (2014); Roberts (2019). This advancement not only facilitates a more precise and comprehensive analysis of energy efficiency but also paves the way for the development of more energy-efficient parallel programs, ultimately contributing to the broader goal of green computing.

Our proposed framework integrates tracing-based and profiling-based measurement techniques to redefine the boundaries of analysis regions. The tracing approach supports a hierarchical region analysis model, allowing users to visualize how the efficiency of internal components affects the program's overall scalability. Meanwhile, the profiling-based approach ensures low-overhead power monitoring with performance counters detailed in Section 3.7 and also intregrating with IPMI and RAPL described in Section 2.4.2, combining detailed insights with minimal performance intrusion. The collected data can be exported as a file, enabling interpretation by other tools to create energy models or to be rendered by graphics libraries for visualization and analysis.

In addition, the framework incorporates an aggregation feature to reduce the volume of data produced while preserving accurate analysis capability with minimal overhead. This mechanism operates by collecting and aggregating measurement data from various regions of an application. It ensures that the aggregated data reflects the overall execution

characteristics, such as maximum runtime, start times, and stop times. This streamlined approach ensures efficient utilization of computational resources, optimizing the analysis process. Furthermore, to enhance user experience and streamline workflow, the tool proposes intuitive usability features. These features, detailed in Section 3.5, automate tedious tasks associated with measurement and analysis, significantly reducing developer effort. By automating the execution and consolidation of data across multiple configurations, manual interventions are circumvented, mitigating the limitations of non-computer-centric analysis approaches.

The following sections describe how the framework was built and used for performance analysis. Section 3.2 presents the related works regarding application profiling and tracing tools. Section 3.3 describes the proposed tool architecture and goals; it explains the main features, usage, and collected data. The experimental results are presented in Section 3.10. Finally, the contributions are summarized in Section 6.2, with an outlook of future works.

3.2. State of the Art Profiling and Tracing Tools

In the pursuit of optimizing parallel programs for high-performance systems, understanding and improving energy efficiency is crucial. As highlighted in the introduction, achieving scalable performance involves efficiently utilizing computing resources such as processors or cores as the problem size or number of concurrent tasks increases. Performance analysis tools are indispensable in this optimization process, as they enable developers to discern execution characteristics, identify energy inefficiencies, and isolate computational behaviors that significantly impact energy consumption. Tools such as Caliper Boehme (2016), HPCToolkit Adhianto (2010), Scalasca Geimer (2010), TAU Malony (2006), Vampir Weber (2019), and VTune Intel (2021) play a critical role by collecting detailed execution data to guide developers in enhancing both performance and energy efficiency. These tools differ primarily in their data measurement and analysis strategies—whether tracing-based or profiling-based, post-mortem or real-time, focusing on individual or comprehensive observation,

and whether they provide visual elements to facilitate data interpretation. Additionally, frameworks like DASHING Islam (2019) and the HATCHET library Brink (2020) integrate performance data from multiple sources to provide a robust dataset for analysis.

Understanding a parallel program's execution sequence as a series of significant activities is crucial for analyzing its behavior Houstis (1997). Events are the basic units of this analysis, and the way they are observed influences data collection strategies.

The profiling-based approach analyzes program performance by measuring metrics such as execution time, CPU utilization, and memory usage. Tools designed for this type of analysis collect information from events during program execution, typically through statistical sampling using interrupts. These interrupts can be triggered by periodic intervals or hardware events, enabling the system to gather relevant data based on the observation focus. Such tools employ statistical techniques to describe program behavior through aggregate performance metrics, often ignoring the chronology of events. This approach is particularly useful for identifying issues like load imbalance, high communication time, or excessive routine calls. Examples of tools using this strategy include Paradyn Miller (1995) and Periscope Ott (2010).

Conversely, tracing-based analysis involves comprehensively collecting information about a program's execution flow, including function calls, data accesses, and communication events. These tools gather performance data from events occurring when the program enters specific states, providing invaluable insights into the temporal aspects of program execution. This method allows users to pinpoint when and where transitions of routines, communications, and specific events occur. However, this approach is generally more invasive and intrusive, often generating significantly larger datasets compared to the profiling-based approach. Examples of tracing tools include Vampir Weber (2019) and Paraver Labarta (2005). Additionally, some tools, such as HPCToolkit Adhianto (2010), Scalasca Geimer (2010), TAU Malony (2006), and VTune Intel (2021), support both profiling-based and tracing-based analysis.

In this work, we propose using tracing-based methods to identify parallel regions in combination with profiling-based methods to measure relevant performance counters for modeling applications. To mitigate some disadvantages of tracing-based approaches, our framework implements

features such as automatic instrumentation and resource aggregation, alongside time-based analysis. Our automatic instrumentation mode is non-invasive, allowing users to analyze an application's execution without modifying the source code or executable.

Runtime overhead refers to the additional instructions executed to collect program measurements, which can vary depending on the tool used and the aspects being measured. This "extra code" incurs time overhead, directly impacting the comprehensiveness and accuracy of the measurements. Excessive overhead can distort the understanding of program behavior and misdirect optimization efforts toward less effective areas. Comparative studies have revealed that traditional tools often impose a runtime overhead ranging from 2% to 40% Eriksson (2016). Thus, minimizing this overhead is paramount for accurate analysis and efficient optimization strategies.

Although many other analysis tools offer automatic instrumentation modes, this work presents a detailed approach to mapping and measuring code parts. Our strategy extends the analysis to parallel regions as a primary objective, effectively identifying the scaling trends of parts or the entire parallel program. By collecting only specific runtime data, we reduce the overhead associated with the measurement process. Similar to tools like HPCToolkit Adhianto (2010) and TAU Malony (2006), our framework supports any version of OpenMP . Despite the program's specific characteristics, our instrumentation strategy results in negligible overhead (less than 1%), making it optimal for analyzing trends in parallel applications, as demonstrated later.

The timing of measurement, data collection, and analysis tasks varies. Real-time analysis continuously monitors performance data during program execution, providing immediate feedback for swift adjustments and optimizations, particularly valuable for diagnosing performance issues in dynamic or time-critical applications. However, many performance analysis tools favor a post-mortem approach.

Post-mortem analysis retrospectively examines collected performance data after program execution to identify inefficiencies and areas for enhancement. This method offers thorough investigation but lacks immediate insights during execution. Examples of post-mortem performance analysis tools include HPCToolkit Adhianto (2010), Scalasca Geimer (2010), TAU Malony (2006), and VTune Intel (2021). These tools may

require storing large amounts of data, but they are best suited to provide an overall view of the execution.

Run-time analysis tools perform both measurement and analysis at run time, accurately detecting waiting states and communication inefficiencies. Examples of such tools include Periscope Ott (2010) and Paradyn Miller (1995). However, the run-time analysis generally requires the coordinated action of tools, increasing structural complexity. The need for synchronous initialization and communication between analysis resources, as well as their impact on the runtime environment, are drawbacks of this approach. Additionally, the scalability analysis, relying on data collected at the end of execution, may be degraded by infrastructure overhead, not benefiting from the characteristics of real-time tools.

Energy management solutions can also provide similar measurement and analysis features. For example, GEOPM Eastep (2017) measures the time and energy of specific regions. An essential contribution of our tool, which we have not found in any other framework, is its ability to provide specific energy consumption data of parallel regions in a fully automated and non-invasive way. For example, in GEOPM, it is necessary to instrument the source code to obtain the same result.

By focusing on objective analysis, we introduce an innovative tool that offers a fresh perspective on observing the scalability trends of parallel programs, addressing the challenges highlighted. Our framework is designed to collect only the essential information needed to infer the overall behavior of the program or specifically chosen parts of it, incorporating features relevant to effective measurement and analysis. Automation is a key component of our approach, enabling the acquisition of comparative measurements across multiple runs. This observability of scalability trends involves correlating program behaviors across various executions with different configurations. Consequently, our framework automates program executions based on user-specified configuration parameters, generates datasets with comparative runtime measurements, and optimizes the volume of produced data. To manage data efficiently, our approach employs aggregation, streamlining storage and facilitating comprehensive analysis, as detailed in the forthcoming sections.

3.3. Framework Architecture

To analyze energy, performance, and scalability trends of parallel programs, we have developed a tool that measures, collects, combines, and compares data from multiple runs. The core of the tool is built around two primary concepts: actuators and sensors. Both actuators and sensors are integrated into the tool as modular components.

Actuators represent parameters that we aim to control. These are variables representing elements external to the program that, when modified, can influence aspects such as performance and efficiency of the running application. Understanding the state of actuators is essential for assessing their impact on program behavior, particularly on scalability and energy consumption. By default, the tool includes actuators that control the number of active cores and threads, program input, and CPU operating frequency.

Sensors are designed to measure and monitor variations resulting from the actions of actuators. Currently, the tool implements three types of sensors:

- Begin/end: collects data at the start and end of each program run.
- Time sample: periodically collects data.
- Event-based: collects data when specific events occur.

Sensors are designed to measure and monitor variations resulting from the actions of actuators. Currently, the tool implements three types of sensors. The *begin/end* sensor collects data at the start and end of each program run. The *time sample* sensor periodically collects data throughout the program's execution. The *event-based* sensor gathers data when specific events occur during the program's execution.

Currently, there are sensors to measure time, energy, and performance counters. The default sensor is a *begin/end* type used to collect the execution time of the entire application. To measure energy consumption, we developed sampling sensors capable of retrieving data from RAPL and IPMI, which are interfaces that provide power information from the CPU and the entire system. Additionally, the *time sample* and *begin/end* sensors gather performance counters data.

Scalability analysis relies on measurements obtained from event-based sensors. To facilitate this, the tool incorporates markers that automatically trigger events, enabling the identification of boundaries within source code parallel regions. These boundaries are typically defined when new threads are created and are well specified in low-level libraries like POSIX, Pthreads, and OpenMP. This capability allows us to automatically hook calls related to the creation and destruction of threads to identify parallel regions and trigger the collection of sensor data directly from the binary code. This seamless integration of sensor functionality into software applications enhances our ability to monitor performance accurately. Additionally, the tool provides manual markers that developers can insert into the source code to monitor specific parts of the program. Furthermore, any system event can be configured as a trigger. The following section outlines the procedure for instrumenting the source code with manual markers.

Fig. 3.1 illustrates the integration of each component within the proposed software architecture. The modular design allows for easy addition or removal of actuators and sensors. The core of the tool manages all operations, including launching the application and gathering data.

3.4. Instrumentation and Intrusiveness

Instrumentation is the process of integrating monitoring or measurement capabilities directly into the software application. Therefore, the instrumentation module, which directly interfaces with the source code (as depicted in Fig. 3.1), emerges as one of the most critical components. Besides automating instrumentation, it determines the level of intrusiveness and overhead associated with the process.

The instrumentation module is designed to execute the fewest instructions possible in the most optimized manner. Currently, it supports C/C++ languages through shared libraries, facilitating both automatic and manual instrumentation of the source code. Manual instrumentation is preferred for cases requiring precise examination of specific code sections.

Manual instrumentation provides three routines for use with the source

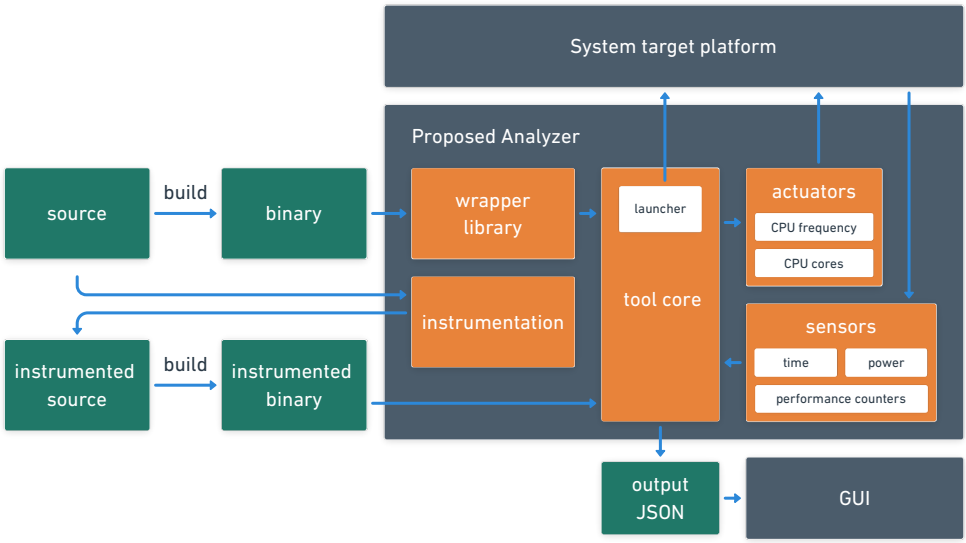


Figure 3.1: Pascal Analyzer architecture showing the interconnections of the central parts of the tool. Whether using a binary (with the wrapper library) or an instrumented source code, the target application can be launched on the target platform by the tool core following the configuration parameters chosen by the user while deploying actuators/sensors. The resulting data collection is stored in JSON file format for post-analysis and visualization (GUI).

code: one for initialization, another to mark the beginning of the region of interest (`pascal_start`), and a routine to mark the end of that region (`pascal_stop`). The initialization routine is called when loading the library to create the necessary data structures and set up data exchange communication. The `pascal_start` and `pascal_stop` routines collect thread identifiers and store timestamps. These routines are implemented to ensure thread safety by allowing only one thread at a time to write in a designated position of a two-dimensional array, thus eliminating the need for locks.

Automatic instrumentation includes a routine that intercepts the creation of threads via the `LD_PRELOAD` environment variable. This routine overwrites parts of an existing native library, such as `pthread_create` in the thread library, GCC implementation `GOMP_parallel`, or Clang implementation `__kmpc_fork_call` in the OpenMP framework. Similar functions for other compilers are intercepted to automatically identify parallel regions. This approach is less intrusive than methods like using a debugger interface with breakpoints or performing binary code instrumentation.

A critical aspect of performance analysis tools, especially when examining real-time program execution, is ensuring that instrumentation minimally impacts program behavior and execution time. As outlined in Section 3.3, our tool supports three types of sensors, each varying in intrusion level. There is minimal intrusion from the `begin/end` sensor, as it only involves the invocation of data collection routines at the start and end of program execution. However, with sampling sensors, the level of intrusion fluctuates based on the sampling rate and the application's total execution time, making intrusion assessment a case-specific endeavor.

The instrumentation overhead does not depend on the number of instructions or the runtime required to process the set of instructions to be analyzed. However, it varies according to the number of processing units used and the measurements taken. To estimate the magnitude of the overhead (T_m), we measured recurring calls to the proposed sampling functions delimiting the regions in a simple benchmark code. This experiment was conducted on the same target architecture described in the experimental results of Section 2.4, and the code structure is presented in Listing 3.1. The time required to execute N calls to the `pascal_start` and `pascal_stop` functions was obtained using the `gettimeofday` routine

from the `sys/time.h` library.

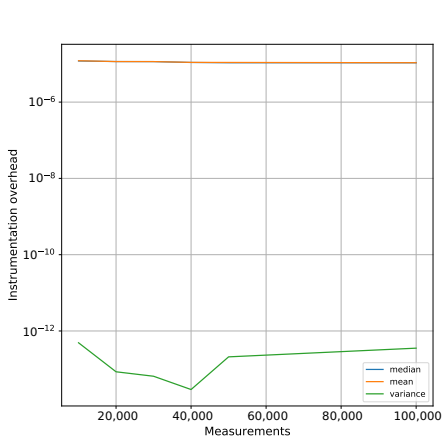
```
// Measure start time
#pragma omp parallel for
for (c=0; c<interations; c++) {
  pascal_start(1);
  usleep(1e4); // to simulate a simple operation
  pascal_stop(1);
}
// Measure end time
```

Listing 3.1: Measuring the Overhead of Instrumentation Functions `pascal_start` and `pascal_stop`, defined as T_m.

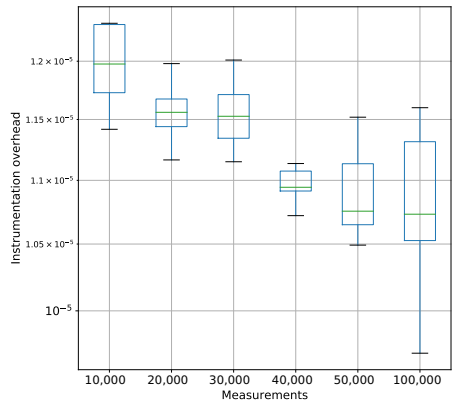
The algorithm above was executed with 1 $\times 10^4$, 2 $\times 10^4$, 3 $\times 10^4$, 4 $\times 10^4$, 5 $\times 10^4$, and 1 $\times 10^5$ calls to the pair of functions, each test repeated ten times. The mean, median, and variance values were computed. Fig. 3.2 shows these results. Fig. 3.2a presents the mean, median, and variance of the time for a single call to our instrumentation function while varying the number of calls/measurements. Fig. 3.2b complements these results, showing the variance in each execution.

Table 3.1 shows the results from the same experiment (Listing 3.1), comparing the time without and with the analyzer (columns Real-Time and Analyzer, respectively). We observe that the proportional impact (overhead) remains constant while the number of iterations increases. Table 3.1 also presents data referring to the simulation using the TAU profiling tool. For this simulation, we replaced the analyzer directives with the time measurement directives (`TAU_PROFILE_START` and `TAU_PROFILE_STOP`) of TAU, allowing us to approximate the measurement and analysis conditions.

From the results, it is evident that the tool proposed in this work has a higher overhead than TAU, but the analysis capabilities are distinct. TAU adds the individual runtime of each thread to define the execution time of a parallel region. This strategy does not account for the simultaneous, specific actions of threads in processing instructions, leading to potential double-counting and inaccurate measurement of the parallel region. The Analyzer addresses this problem by counting intersection periods only once. Therefore, despite TAU's lower overhead, our more accurate measurement



(a) Time consumed from sampling a region one time in seconds.



(b) Box plot showing the statistics of the sampling time of a single region.

Figure 3.2: Measuring variance of the time to single instrumentation, i.e., a call to `pascal_start` and `pascal_stop` while varying the number of measurements.

Table 3.1: Instrumentation overhead estimation varying the number of samples collected with TAU and Analyzer.

Iterations	**Time (s)**			**Overhead (%)**	
	Real Time	**TAU**	**Analyzer**	**TAU**	**Analyzer**
10,000	100.933	100.992	101.053	0.058	0.118
20,000	201.863	201.980	202.094	0.057	0.114
30,000	302.797	302.977	303.142	0.059	0.114
40,000	403.738	403.978	404.176	0.059	0.108
50,000	504.675	504.959	505.212	0.056	0.106
100,000	1009.359	1009.927	1010.432	0.056	0.106

is crucial for effective scalability analysis.

Varying the number of threads impacts the cost of instrumentation. As shown in Table 3.2, the overhead percentage tends to rise with the application's runtime. However, this increase becomes negligible when considering the exponential growth in thread count.

Table 3.2: Instrumentation overhead while varying the number of threads with the number of samples fixed to one million.

Threads	**Time (s)**		**Overhead (%)**
	Direct	**With Analyzer**	
1	1009.360	1010.430	0.107
2	504.733	505.374	0.127
4	252.399	252.742	0.135
8	126.215	126.390	0.138
16	63.109	63.199	0.143

Increasing the processing load favorably affects the relationship between execution time and overhead percentage. As processing demand grows, runtime naturally increases, while the overhead changes only with the number of threads.

For pluggable sensors, overhead is generally not a concern as they run on a separate thread with minimal CPU usage. However, specific scenarios may cause interference. One such scenario is when the application requires all the machine's resources simultaneously, and sensors respond faster than the processing speed at a given sample rate. In this case, the overhead is directly linked to the sample rate and system concurrency management. Such occurrences are rare in HPC, as they require an application with perfect linear scaling. Another potential scenario is I/O overhead, where network, disk, or memory resources may become unavailable due to high sensor usage. However, this scenario is even less common, as sensors seldom produce data that quickly; in all the tests performed, this was not an issue.

3.5. Features and Usage

The analyzer is a simple and easy-to-use tool designed to help users understand the general behavior of a program before investing in more in-depth and long-term analysis. To meet various needs and resource constraints, the tool offers several functionalities that allow users to parameterize the measurement process. Key features include:

1. Automatic binary instrumentation for OpenMP and Pthreads applications, with optional manual source instrumentation for more precise control.

2. CPU actuators for controlling cores, frequency, and enabling/disabling hyperthreading. Application actuators to automatically execute with multiple inputs.

3. Sensors based on performance counters enabling measurement of instructions executed, cache misses, , and more. Integration with RAPL and IPMI for precise power measurements.

4. Aggregation of collected metrics by region, significantly reducing disk and RAM usage. Users can choose to group the data by mean, median, minimum, or maximum values.

5. Hierarchical regions: facilitating the identification of aligned regions and enabling the analysis of the calling hierarchy of inner regions and blocks.

The tool can be used via the command line or through its API. The API provides calls to integrate sensors and actuators, as shown in Listing 3.2.

```
1 from analyzer.run import Run
2 from analyzer.actuators import CPUFrequencies
3 from analyzer.actuators import CPUCores
4 from analyzer.sensors import RAPL
5 from analyzer.sensors import fingerprint
6 from itertools import product
7
```

```
lsensors = [
    RAPL(),
    fingerprint(counters=["INSTRUCTIONS"])
]
cores = [
    CPUCores(c) for c in range(1, 32)
]
freqs = [
    CPUFrequencies(2800000),
    CPUFrequencies(2600000)
]

# all combinations
configs = list(product(cores, freqs))
app = Run(application="a.out",
          repetitions=10,
          instr_auto=True,
          sensors=lsensors)
app.run(configs)
app.savedata("out.json")
```

Listing 3.2: Python script showing some API features provided by the tool, and how a custom run can be configured.

3.6. Exported Data Structure

The data collected by the tool is exported to a JSON file and stored on disk. This data structure is divided into two main groups: configuration parameters and sensor measurements. The configuration encompasses information about the host machine, sensors, and actuators and their configuration. The data is organized by keys representing a unique combination of actuators derived from the actuator's value.

Listing 3.3 provides an example of data collected by the analyzer exported to a file. The file consists of several main sections, starting with a header containing essential system information. Following the header, the description of the collected data is presented, which includes

a list of actuators and the specific types of information collected from the sensors. The collected samples are then presented, and organized by actuator configurations and sensor types.

```
{
  "config": {
    "execdate": "Execution timestamp",
    "kernel": "Kernel version or system details",
    "pkg": "Package or configuration name",
    "data_descriptor": {
      "keys": [ "Parameter1", ..., "ParameterN" ],
      "values": [ "MeasuredMetric" ],
      "extras": {
        "sensors": {
          "values": [ "SensorType1", ..., "SensorTypeN" ]
        }
      }
    },
    "arguments": [ "Input parameters for execution", ... ]
  },
  "data": {
    "key1;key2;key3;keyN": {
      "MeasuredMetric": "Value of the metric",
      "sensors": {
        "SensorType1": [ [ "SensorReading1", "Timestamp1" ], ... ],
        ...
        "SensorTypeN": [...]
      }
    },
    ...
  }
}
```

Listing 3.3: Sample exported data file showing the internal structure and organization of the data.

The keys and values in the JSON file might not be easily identifiable and understandable for users. However, they can be efficiently interpreted by a script or visualization software such as PaScal Viewer Silva (2018).

The PaScal Viewer provides a visualization module that interprets and presents the data in an organized and user-friendly manner, enhancing the usability of the collected data.

3.7. Profiling with Performance Counters

The PaScal Analyzer utilizes performance counters to gather essential data on application behavior, crucial for optimizing performance and energy efficiency in High Performance Computing (HPC) systems. These counters, embedded within modern processors, count micro-architectural events such as instructions executed, cache hits, branch mispredictions, and energy estimations. This capability is integral to understanding the intricate dynamics of how applications utilize system resources, offering unprecedented insight into application performance and system utilization.

Originally developed for debugging, performance counters now play a pivotal role in diverse applications including software profiling Melo (2010); Kufrin (2005); Knupfer (2011), CPU power modeling Zamani (2012), DVFS, malware defense Demme (2013), and more. Their ability to capture real-time, granular performance data without impacting application execution makes them indispensable for any in-depth analysis of HPC systems.

This profiling module provides a methodological approach for collecting and analyzing these metrics, enhancing our capacity to discern energy consumption patterns and computational bottlenecks. It combines a high-level Python API with an efficient C++ backend, ensuring that the integration of performance counter data collection into application analysis is both user-friendly and minimally invasive. This balance is vital for maintaining the integrity of performance data while ensuring minimal overhead.

Moreover, exploiting Performance Monitoring Units (PMU) effectively necessitates a deep understanding of both micro-architecture and kernel APIs, along with strategies to manage their growing complexity. Despite the availability of tools leveraging performance counters, high-level, accurate programmable interfaces are scarce. Most existing APIs, such as PAPI Weaver (2013); Mucci (1999) and Perfmon Eranian (2008), face challenges related to poor documentation, instability, or narrow design focus, which limits their usability and precision.

Our approach addresses these challenges by providing efficient, low-level access to PMUs complemented by high-level, user-friendly programmable interfaces. This strategy allows for comprehensive configura-

tion and post-processing via Python, while the underlying C++ module handles precise data collection, thus minimizing overhead and ensuring measurement fidelity. The accuracy of this module has been rigorously tested against existing tools, confirming its superior capability in capturing and processing performance data, thereby facilitating effective application fingerprinting and subsequent optimization strategies.

3.7.1 State of the Art Performance Counters APIs

There are only a few APIs allowing access to performance counters. PAPI Weaver (2013); Mucci (1999), one of the most used libraries for accessing hardware performance counters, was originally developed to provide portable access to the counters found on a diverse collection of modern microprocessors. Instead of creating a new performance framework from scratch each time it is adapted to a new machine, developers can write measurement code using the PAPI API, which abstracts the underlying interface.

PAPI was developed on C and a few non-official libraries were ported to Python. The main problem with the Python versions, besides having a considerable overhead, is that it does not have an easy way to create raw events, or low-level control, without having to use a special driver. And, as our tests will show later, counter-sampling over time does not produce good results either.

There are also available sets of interfaces using their drivers, mainly because the counters are only accessible in kernel mode (ring 0) to control the events for which the counter must be started or stopped. As such, the hardware performance counters are only accessible by the operating system's kernel, which operates at the highest privilege level. Some events are fixed and others require the development of a dedicated kernel driver. Perfctr Nikolaev (2011) supports per-kernel-thread and system-wide monitoring for most major processor architectures. It is distributed as a stand-alone kernel patch. The interface is mostly used by tools built on top of the PAPI performance toolkit. The Intel VTUNE Intel (2021) performance analyzer comes with its kernel interface, implemented by an open-source driver. The interface supports system-wide monitoring only and is very specific to the needs of the tool.

The problem with the approach of a tool and its kernel interface

is dangerous because, as mentioned on Eranian (2008), there is clearly code duplication, but more importantly, there is no coordination between the various interfaces that may coexist sharing access to the same PMU resource. To solve this problem, Perfmon2 Eranian (2008) offers a standard interface that all tools can use. Unfortunately, it has not been widely adopted, just supported by a few architectures like the IA64. Instead, Linux comes up with a performance counters subsystem which provides a complete set of configurations.

Despite the availability of various APIs, there remains a need for standardized interfaces and improved coordination among tools accessing PMUs. While Linux's performance counters subsystem offers a comprehensive set of configurations, widespread adoption of standardized interfaces like Perfmon2 remains limited.

3.8. Reading Performance Counters Precisely

Our performance counter module is architecturally segmented into five primary blocks: Profiler, Events, Workload, Analyzer, and libpfm4, spanning across two operational levels: interface and kernel. This structure establishes a robust framework for performance monitoring, adept at handling complex data interactions and ensuring reliability in data acquisition and analysis.

Figure 3.3 illustrates the interconnected roles of these blocks within the tool's architecture.

This integrated system design ensures that our tool not only efficiently reads and processes performance counters but also ensures the data's accuracy and actionability for performance optimization in high-performance computing environments. Detailed descriptions of each module's functionality are provided below.

3.8.1 Workload Module

The Linux system provides a performance monitoring subsystem that facilitates access to per-task and per-CPU performance counters, counter groups, and related event features via an abstraction layer over the hardware capabilities. These counters are managed through Model-Specific

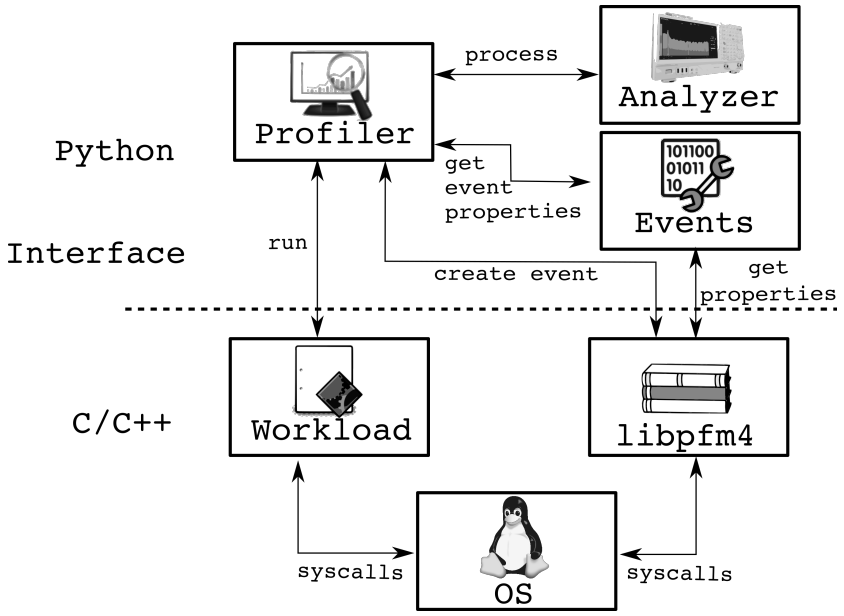


Figure 3.3: The Performance Counters Module Interconnection Structure

Registers (MSR), which are accessible only in kernel mode (ring 0).

All events are modeled as 64-bit virtual counters, regardless of the underlying hardware counter widths. They are accessed through special file descriptors, with each descriptor dedicated to one virtual counter. These counters can be managed via interrupt handling, polling, or time-based sampling. Interrupt handling involves a user-defined function triggered by an event such as a counter overflow, leading to a signal that passes control to the handler function. Polling allows reading from a queue of events processed by the system, and time-based sampling involves reading the counters at specified intervals.

The Workload Module, developed in C++, is the core of our tool, orchestrating application deployment and managing the sampling process. Key functionalities include synchronization at application start, halting execution before the first instruction, and utilizing the debug interface on Linux to ensure counters are reset prior to application launch. This module offers extensive control over the runtime system through seamless interfacing with Python via the Python C API.

3.8.2 Profiler

The Profiler serves as the user interface for selecting and configuring monitoring events. Given the hardware constraints where only a limited number of counters are available at any given time, events must be carefully selected and, if necessary, multiplexed, sacrificing some precision for broader coverage. The available counters differ by processor architecture; for instance, modern Intel CPUs Intel (2013) support three fixed and four programmable counters per core, monitoring events like Instructions Retired, logical cycles, and reference cycles.

3.8.3 Events Module

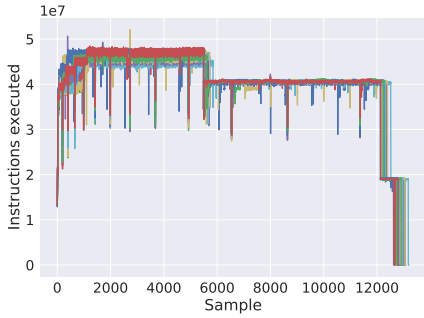
The Events Module catalogs a wide array of event parameters, configurations, and the available PMUs. It handles system calls for creating and reading these events, mostly managed directly by the Workload Module or indirectly via an intermediary Python link to the libpfm4 Eranian (2008). The libpfm4 library is crucial for translating event names (specified as strings) into their respective hardware or OS-specific codes, which is essential for configuring the monitoring setup.

3.8.4 Analyzer Module

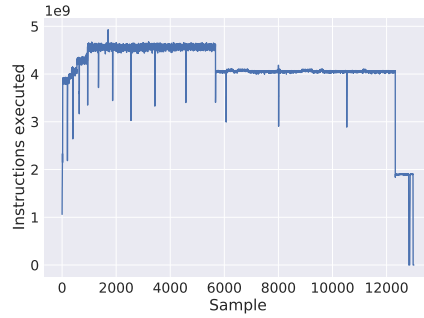
Despite expectations, hardware performance counters often do not yield exact and deterministic results, evidenced by run-to-run variations and overcounts, especially in x86-64 machines Weaver (2008, 2013); Das (2019); Guire (2009). This variability poses challenges for applications requiring strict determinism, such as deterministic replay or threading libraries. To address these issues, we have developed a methodology to minimize noise and overcounts using multiple application runs.

The Analyzer Module plays a critical role in post-processing the data collected. It applies various techniques to refine the data, such as median filtering to remove outliers, interpolation to standardize data points, and application of Savitzky-Golay filters for smoothing. This comprehensive suite of functions facilitates comparative analysis by reducing noise and interpolating values to present a clear, consistent dataset for further analysis.

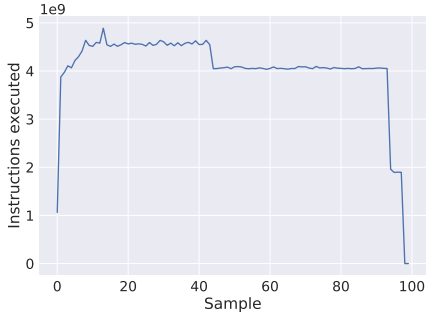
The post-processing phase is graphically represented in Fig. 3.4, which illustrates the transformation of raw execution data into a refined format ready for analytical assessment.



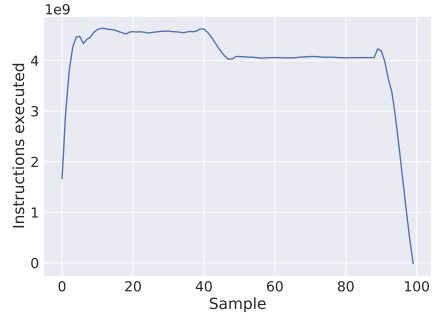
(a) Input executions curves with different inputs.



(b) Single execution representation after median filtering.



(c) B-spline interpolation of the curve showing normalized sample number.



(d) Final curve after applying Savgol filtering and normalizing.

Figure 3.4: Post-processing workflow steps applied to the Polybench 2mm application, depicting the process from raw data collection to final smoothing for analysis. Each subfigure illustrates a key step in the data refinement process.

The steps include:

- **Median Filtering:** Each set of run data undergoes a median filter to remove outliers and reduce variability, helping to clarify the underlying trends.

- **Interpolation:** The median-filtered data is then interpolated using a B-spline algorithm Hang (2017), which ensures that all curves have the same number of points for consistent comparison.

- **Normalization:** This step normalizes the horizontal axis over the interval from 0 to 100, allowing the comparison of program execution in a consistent interval, rather than over time.

- **Savgol Filtering:** Finally, we apply a Savitzky-Golay (Savgol) filter Luo (2005) to smooth the data further, enhancing the signal-to-noise ratio without significant distortion.

3.9. Performance Counters and Intrusiveness

Another important aspect to validate is the intrusiveness of our performance counter module. For that, we compared the results of the counters obtained with different APIs. We used the hand-crafted assembly benchmark from Weaver (2013), designed to test the determinism and accuracy of PMUs. We compared the values obtained from the Linux API, PAPI on C, and PAPI on Python. The events used for this comparison were instructions retired, branch instructions, memory read, memory load, and arithmetic operations. We ran the benchmark 30 times and calculated the mean and standard deviation as shown in Table 3.3. Some events could not be measured using PAPI because the tool does not accept raw events and there are no equivalent events.

Table 3.3: Comparison with other state-of-the-art APIs with our proposed approach

	Average$\times 10^{-6}$					**Standard deviation**			
Counters	Pined values	Linux API	PAPI	PAPI Python	**Proposed**	Linux API	PAPI	PAPI Python	**Proposed**
INSTRUCTIONS_RETIRED	226.99	227	227	225.9	227	396	133	337763	175
BRANCH_INSTRUCTIONS_RETIRED	9.24	9.25	9.25	9.24	9.25	297	208	8485	91
BR_INST_RETIRED:CONDITIONAL	8.22	8.22	8.22	8.21	8.22	0	0	3383	0
MEM_UOP_RETIRED:ANY_LOADS		2484.18			2484.16	37399			38953
MEM_UOP_RETIRED:ANY_STORES		189.96			189.96	1513			687
UOPS_RETIRED:ANY		12291.08			12290.9	345246			333298
PARTIAL_RAT_STALLS:MUL_SINGLE_UOP		0.6			0.6	1222			521
ARITH:FPU_DIV		5.8			5.8	1760			1544
FP_COMP_OPS_EXE:X87		48.79			48.79	1283			3311
INST_RETIRED:X87		17.2			17.2	4			3
FP_COMP_OPS_EXE:SSE_SCALAR_DOUBLE		5.4			5.4	1547			2097

Since the benchmark was hand-crafted with assembly, we know exactly the value for some counter events. For this reason, the number of instructions, branch instructions, and conditional branches are pinned. However, some other events are architecture-specific and there is no pined value. In the latter case, we can still compare to the Linux API, which should be closest to reality.

The differences using the Linux low-level API, PAPI, and our proposed tool on Table 3.3, are negligible (the average percentage difference is less than 0.01% in all the cases). As expected, PAPI on Python had the largest difference (with an average difference of 0.25%) mainly due to an unsynchronized start that resulted in the loss of some instructions at the beginning of the execution. This can be an important problem if the application contains a small number of instructions.

The standard deviation of the 30 executions shows that our tool has the smallest variation on a run-to-run on most events. On the contrary, PAPI on Python shows a big variation compared to the others.

3.10. Pascal Analyzer Use Cases

In this Section, we discuss the measurements and simulation results for three distinct cases, including two real applications. We also demonstrate how external libraries and standalone visualization tools can be used to render the collected data and investigate performance aspects such as the program's scalability capacity and energy consumption.

We used three experiments to assess the tool and demonstrate its ability to support analysis aimed at observing parallel scalability. The first includes a runtime imbalance between processing units in a specific parallel region. In this case, the objective is to present how the analyzer helps users observe the impact of the inefficiency of a code part on the whole program. The code for this experiment is presented in Listing 3.4. It consists of two simple parallel regions with the same functionality that divides the iterations of a loop between the available threads. The difference between the regions lies in the strategies used to manipulate the sum variable, used in the example to store the values of the calculations performed in each thread. We assume that the `anyop()` function invariably has the same runtime in all function calls.

```
1  sum = 0;
2
3  #pragma omp parallel for schedule(static) reduction(+: sum)
4  for (i = 0; i < operations; i++) {
5    sum += anyop();
6  }
7
8  #pragma omp parallel for schedule(static)
9  for (i = 0; i < operations; i++) {
10   #pragma omp critical
11   sum += anyop();
12 }
```

Listing 3.4: Sample code used to visualize the impact of 2 regions on program scalability.

The first directive on line 3 instructs the compiler to parallelize the following for loop using OpenMP. The `parallel for` clause indicates

that the iterations of the for loop should be executed in parallel. The `schedule(static)` clause specifies that the iterations of the loop are divided into equal-sized chunks, with each thread executing a chunk. Static scheduling is typically used when the workload of each iteration is uniform. The `reduction(+:sum)` clause specifies a reduction operation on the `sum` variable. The `+` symbol indicates that the sum of all the private copies of the variable `sum` (created for each thread) should be computed and combined at the end of the parallel region. This ensures that the final value of `sum` is the total sum of all iterations, thus avoiding race conditions.

The second directive on line 8 also tells the compiler to parallelize the following for loop using OpenMP with static scheduling. However, this loop contains a critical section. The `critical` directive ensures that the code inside the critical section is executed by only one thread at a time. This prevents race conditions but can lead to performance bottlenecks due to the serialization of the critical section.

The two regions in this code are used to compare the scalability of parallel reductions versus the use of critical sections. The first region demonstrates a scalable reduction operation, while the second region shows a potential bottleneck with critical sections.

The command presented on Listing 3.5 was used as the base to perform experiments. For the experiment of Listing 3.4, we use just the parameters `-c`, `-r`, `-t`, `-a` and `-o`, including the 64 value to `-c` option.

```
1  analyzer
2  application
3  -c 1,2,4,8,...,32 # threads/cores
4  -r 3 # number of repetitions
5  -t auto # automated instrumentation
6  -a 1 # aggregation mode
7  --ipmi ip user psswd # energy sensor
8  --idtm 5 # idle time between runs
9  --dhpt # disable hyper-thread
10 --dcrs # disable cores
11 --ipts ... # application specific inputs
12 -o application.json # output file
```

Listing 3.5: Command line showing how experiments were run through a terminal.

The command on Listing 3.5 returns a .json file with all the information necessary for our analysis. We can quickly generate tables with the collected data from this file, thus supporting observation and analysis of scalability, energy trends, and model fitting. The code described in Listing 3.6 and Table 3.4 are samples of how users can easily read and view collected data from the control terminal in a tabular way.

```
from analyzer import Data

data = Data("application.json")
data.energy(regions=True)
```

Listing 3.6: Example of using the Python API to load analyzer files.

Table 3.4: Dataframe generated automatically from collected samples using the Python API.

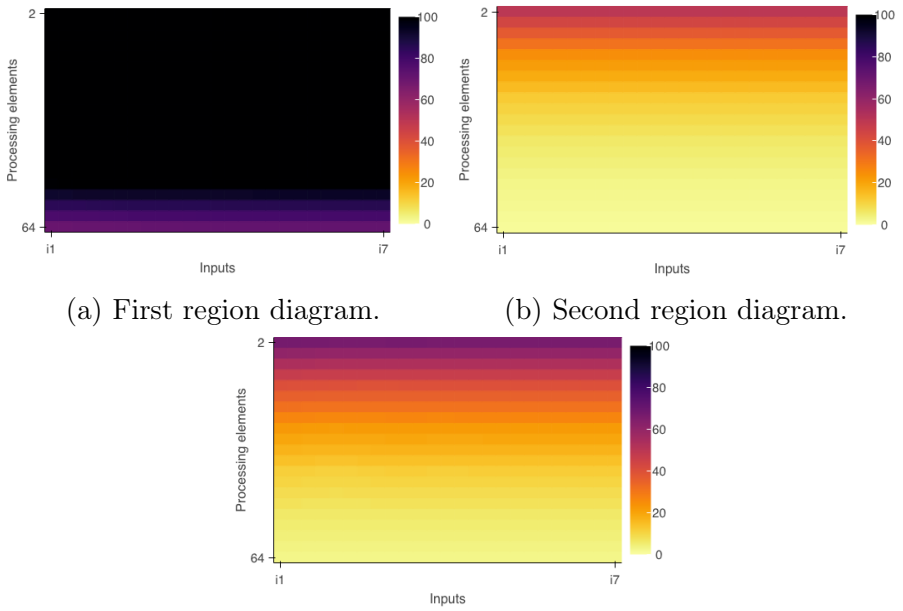
Repetition	Input	Cores	Regions	Start Time (s)	Stop Time (s)	ipmi Energy (J)
1	1	1	1	0.00	66.31	13,156.19
1	1	1	2	0.00	66.27	13,148.87
...	...	...	...	...	...	...
1	5	30	4	0.00	8.46	1903.82
1	5	30	5	0.02	58.16	13,067.01

3.10.1 Case of study

The analyzer does not display graphics and other visual elements natively. However, the simple use of external libraries allows generating graphics and visualizing points of the program's behavior that you want to observe. In addition, specialized visualization tools can also be used to view the results. PaScal Viewer Silva (2018), for example, natively interprets the analyzer's output files, complementing its functionality and creating an integrated and appropriate environment for the program's parallel scalability analysis.

We used PaScal Viewer to observe in a hierarchical form the experiment presented in Listing 3.4. This view allows us to evaluate how the different OpenMP clauses impacted the region's efficiencies. The efficiency, defined as the ratio of speedup to the number of processing units, pinpoints how a program can take advantage of increasing processing elements on a parallel region. PaScal Viewer displays an efficiency diagram for each analysis region, as presented in Fig. 3.5. The x-axis (i1..i7) corresponds to different data inputs and the y-axis (64..2) to the number of threads used in the processing. Comparing these diagrams, it is possible to see how a critical clause damages the scalability. In addition, it is also possible to observe that the second region affects the efficiency of the whole program due to the code characteristics.

If the reduction clause replaces `#pragma omp crtical` (line 11 on the second region of Listing 3.4), the second region and the entire program become more efficient, as shown in Fig. 3.6.

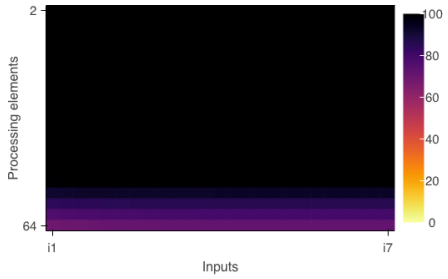


(a) First region diagram.

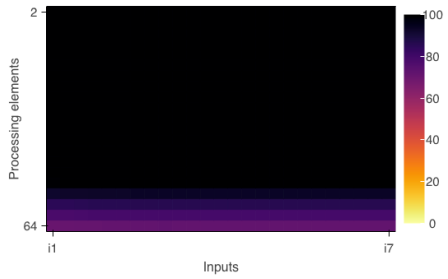
(b) Second region diagram.

(c) Whole program diagram.

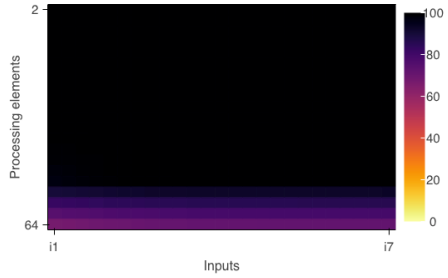
Figure 3.5: Efficiency diagrams and impact of inner regions on program scalability. (x-axis = inputs; y-axis = number of threads).



(a) First region diagram.



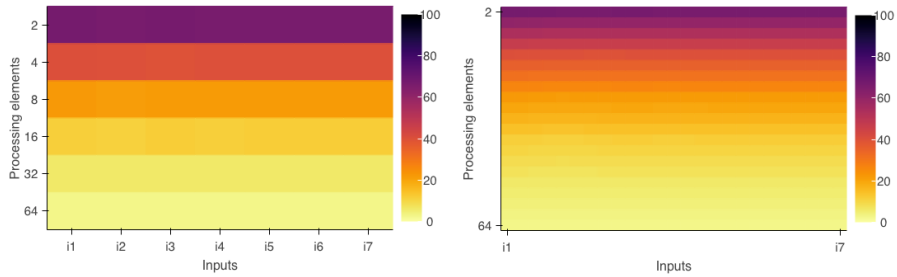
(b) Second region diagram.



(c) Whole program diagram.

Figure 3.6: Efficiency diagrams after removing `#pragma omp crtical` clause. (x-axis = inputs; y-axis = number of threads).

Figures 3.5 and 3.6 were rendered using a tool feature that smoothes the color transition of diagrams. This feature uses interpolation to create a visual element where the color transition is less pronounced. The diagram axes only show the initial and final values with this option. The user can visualize diagrams with only discrete values or even compare the two presentation modes side by side. Figure 3.7 shows the difference between discrete and smoothed modes considering the whole program and the simulation scenario that uses the clause `#pragma omp crtical`.



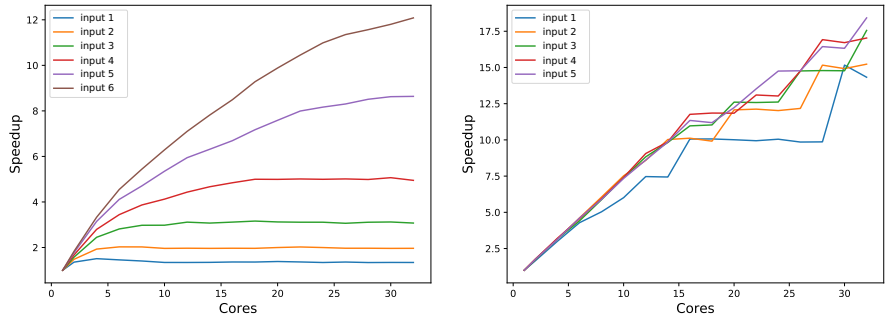
(a) PaScal Viewer diagram on discrete mode

(b) PaScal Viewer diagram on smoothed mode.

Figure 3.7: Visualization modes of diagrams provided by PaScal Viewer.

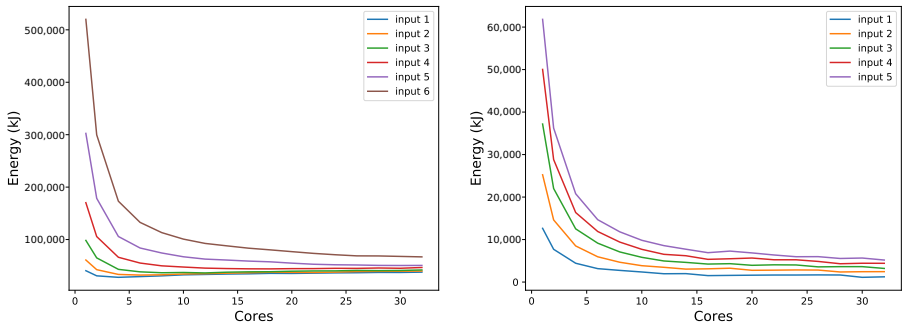
Other analysis objectives not supported by PaScal Viewer, such as visualization of the speedup curve or energy consumption, can be supported by traditional plots. Figures 3.8 and 3.9 present the charts rendered for analysis of the Raytrace and OpenMC applications. In these figures, it is possible to observe that the program efficiency varies according to the increase in the number of threads and with higher processing loads (execution of inputs).

From the diagrams in Figure 3.10, it is possible to observe that the applications exhibit different behaviors concerning their efficiency variations and scalability capabilities. OpenMC maintains its efficiency almost invariant. This pattern represents strong scalability and indicates that the program can maintain its efficiency level when it uses a larger number of threads and processes larger inputs. On the other hand, Figure 3.10a demonstrates that Raytrace achieves higher efficiency values when processing larger inputs. However, it is also possible to observe that increasing the number of processing units fixing the input size will not



(a) Raytrace speedup. (b) OpenMC speedup.

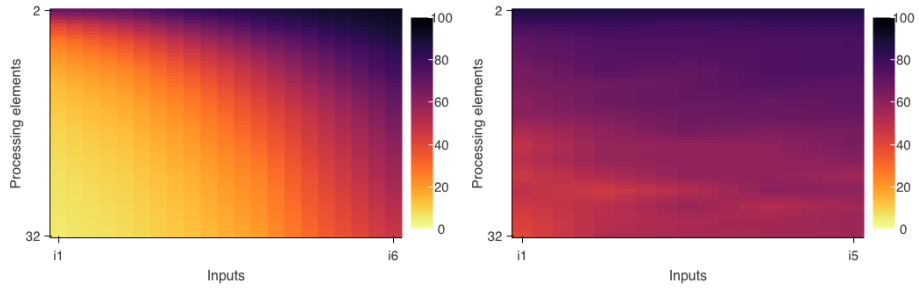
Figure 3.8: Speedup curves for Raytrace and OpenMC applications across different input sizes. These plots illustrate how much they can scale with the number of threads when varing the input size.



(a) Raytrace energy consumption. (b) OpenMC energy consumption.

Figure 3.9: Energy consumption identified in the execution of applications while varying the number of cores for several input sizes. Those plots give an indication of the energy consumption when increasing the number of cores.

improve or hold the efficiency. In the second Figure 3.10b the efficiency is practically constant while varying the number of cores and the input indicating a strong scalable program.



(a) Raytrace efficiency diagram. (b) OpenMC efficiency diagram.

Figure 3.10: Efficiency diagram varying the input size and the number of cores. The color bar indicates the efficiency value in percentage.

The input values are shown in Figure 3.8 to Figure 3.10 indicating different data sets for processing. For example, the input size i2 represents a load that will require sequential processing with runtime twice as long as input i1. Likewise, the input i3 represents a load that will require sequential processing with runtime twice as long as input i2, and so on.

Even if the analyzer does not display graphics natively, analysis aimed at observing scalability and energy consumption of applications depends on precise measurements, reinforcing the advantage of the analyzer proposed in this work.

CHAPTER 4

Application Energy and Performance Models

This chapter introduces analytical models designed to predict and optimize software configurations for energy efficiency in single-node HPC applications. Using the developed framework detailed in Chapter 3 we develop models that leverage a detailed understanding of architecture and application behavior to enhance the effectiveness of DVFS and DPM strategies.

Key parameters, such as the degree of parallelism and dynamic power characteristics, are integrated into the models to predict energy consumption accurately under various hardware configurations. This enables more informed decisions during software development and operation for energy efficiency.

We present a comprehensive modeling approach building on our published work Silva (2018, 2019, 2020) for 13 parallel applications, aiming to identify energy-optimal configurations across diverse input sizes. Our findings indicate potential energy savings of up to 70% compared to default Linux power management settings, with an average improvement of 14%. Additionally, we benchmark our analytical model against conventional machine learning techniques, demonstrating that our model achieves comparable accuracy with approximately tenfold reduction in energy overhead.

4.1. Introduction to Models

While substantial research has been conducted on DVFS, the primary focus has remained on consumer electronics and laptop markets. The concept of energy awareness in HPC systems is is not entirely new Feng (2003). Additionally, the operational characteristics of those systems differ significantly.

Firstly, the workload on consumer systems is highly interactive with end-users, whereas HPC platforms typically handle non-interactive workloads. Secondly, activities on non-HPC platforms often share machine resources, whereas in HPC, each job usually runs with dedicated resources. Thirdly, HPC systems are generally much larger making it more challenging to gather information, organize, and execute global decisions efficiently. Given these differences, it is crucial to investigate whether DVFS scheduling algorithms, effective in conventional computing environments, are equally efficient for HPC systems.

This chapter proposes a comprehensive energy model based on CPU frequency and the number of cores. The model aims to understand and optimize the energy behavior of parallel applications in HPC systems by considering application parameters, such as the degree of parallelism, and CPU parameters related to dynamic and static power. Unlike existing models, the proposed model integrates both frequency and core count in a single equation to estimate the energy consumption of a specific application in a given configuration.

Additionally, by incorporating frequency and active cores, this model can serve as a foundation for DVFS and DPM optimization strategies. It can also analyze the contribution of each parameter (e.g., level of parallelism) to the overall energy consumption. The number of cores is particularly crucial in HPC environments, where applications are designed to run on multiple cores.

The proposed energy model combines an application-agnostic power model with an architecture-specific application performance model. The power model is based on the power draw of CMOS logic gates as a function of frequency Sarwar (1997); Butzen and Ribas (2007) and extends this to include the number of cores. The performance model is rooted in Amdahl's law Amdahl (1967); Eyerman (2010); Gustafson (1988), which

estimates runtime in multi-core systems. This model has been further extended to account for execution frequency and input size, allowing it to characterize application performance on the target architecture.

The main contributions of the proposed model are:

- **Simpler to Compute:** The model is designed to be simple and computationally efficient, unlike complex machine learning models. This simplicity allows for quicker fitting and computation, making it highly effective for DVFS and DPM optimization without requiring extensive computational resources.
- **Parameters with Physical Meaning:** Each parameter has a clear and direct physical interpretation, helping to understand the specific contributions of each factor to the overall model.
- **Analytical Analysis:** The presence of a closed-form equation allows for extensive analytical mathematical analyses, enabling a deeper understanding and exploration of the model's behavior and potential optimizations.
- **Controllable Variables:** The model equation incorporates parameters that can be directly controlled by the system such as frequency, enhancing the precision of optimization efforts.

4.2. Energy Models Theoretical Background

A model is a formal representation of a natural system. The representation of computer system models includes equations, graphical models, rules, decision trees, representative collections of examples, and neural networks. The choice of representation affects the model's accuracy, as well as its interpretability Seel (2012); Roy (2019); Zhu (2019). Accurate energy and power consumption models are essential for many energy efficiency schemes employed in computing equipment Rivoire (2007), and they can have multiple uses, including the design, forecasting, and optimization of data center systems. This study focuses on analytical models that could aid energy optimization and analyses of crucial factors in the total energy draw.

The desirable properties of a full-system model of energy consumption include accuracy, speed, generality and portability, inexpensiveness, and simplicity Rivoire (2008). However, modeling an HPC system's exact energy consumption behavior is not straightforward, either at the whole-system level or at the level of individual components. Indeed, in addition to the type of applications, data center energy consumption patterns depend on multiple factors, such as hardware specifications, workload, or even cooling requirements. Unfortunately, some are not easily measurable, although we can obtain the most relevant ones for applications. Furthermore, it is impractical to perform detailed energy consumption measurements of lower-level components without incurring additional overhead.

Several proposed models have already been classified concerning their input parameters, as shown by Dayarathna et al. Dayarathna (2016), who analyzed more than 200 models according to their characteristics and limitations and classified them where the model is more suited to its objectives. The most relevant categories include system utilization or workload, frequency, and system states (such as cache miss, branch prediction, number of instructions executed and others).

Often, energy models are described as a combination of two main parts, the power model of the system and the performance model of the application. This is because the concept of energy (E) is the total amount of work performed by a system over some time (T), while power (P) is the rate at which this work is performed. In other words, power is the measure of how quickly the system executes the application. The relation between these three amounts can be expressed as:

$$E = \int_0^T P(t)dt. \tag{4.1}$$

In this section, we begin by looking in detail at the power and performance models for systems and applications.

4.2.1 Power Models

The modeling of system parameters is becoming popular nowadays with the advantage of performance counters provided by the CPU or the operating system. These counters can measure micro-architectural events,

such as instructions executed, cache hits, miss-predicted branches, and more; Thus, providing a base for many different estimations of power usage. This makes this type of model very suitable for power estimation because it can use information about several internal states of the computer.

Frequency-based models are the foundation for many power models using performance counters Sarwar (1997); Butzen and Ribas (2007); Usman (2013).

By characterizing the predictable digital circuit behavior with reliable performance measurements, the model can accurately estimate overall system power consumption. One of the most common frequency-based model approximations are defined as follows:

$$P = \alpha + \beta f^3, \tag{4.2}$$

where α and β are model parameters, and f is the operating frequency (details of this equation are covered in Chapter 4).

This type of model is suitable for optimization problems, because in these the operating frequency can be easily controlled.

4.2.2 Performance Models

The most common way to model the application performance is by analyzing its workload. The workload is an abstract representation of the amount of work performed by an application for a given time and speed.

The workload, denoted as W, can be defined in many different ways. A frequently used definition, employed in studies such as those by Paolillo et al. Paolillo (2018), Francis et al. Ishfag (2012), and Kim et al. Kim (2015), is the following:

$$W = \int_0^\tau s(t)dt = s\tau, \tag{4.3}$$

where τ is the total active time, and s is the execution speed in instructions/second.

Utilization models, defined as the ratio between the time that the system is active (meaning when the processor was executing instructions) and the total time (idle and active), are found alternately in the literature Fu (2018); Ishfag (2012). These models are present in many DVFS

algorithms available in Linux. They can be viewed as a good alternative to the workload since it is impossible to measure workload in real time. Eq. (4.4) defines the workload in terms of CPU utilization (u):

$$u = \frac{\tau}{T} = \frac{W/s}{T}, \tag{4.4}$$

where T is the total execution time (idle and active), and τ is the active time.

Models based on CPU utilization are the basis for DVFS algorithms. Even though this is not a controllable parameter, it is straightforward to measure system utilization with almost no overhead, and it is also very portable in terms of operating systems and architectures.

4.3. Energy Models Related Work

Various modeling approaches have been developed to understand and predict the behavior of applications and hardware under different conditions. This section reviews several key models related to energy consumption and performance, highlighting their methodologies and limitations. By examining these different modeling approaches, this section provides a broad overview of the current state of energy and performance modeling, setting the stage for the introduction of a more integrated and application-aware model.

Event-based models have been proposed because supported by some operating systems, they are used to identify the different activities carried out by the architecture during the execution of applications. Merkel et al. Merkel (2006) and Roy et al. Roy (2013) provide foundational examples of such models, calculating total energy based on the frequency and type of events.

Merkel et al. Merkel (2006) developed an energy model for processors based on events. Their model assumes a fixed energy consumption α_i for each activity, and by counting the number of occurrences c_i of every activity, they estimate the total energy as:

$$E = \sum_{i=1}^{n} \alpha_i c_i. \tag{4.5}$$

Another event-based model, introduced by Roy et al. Roy (2013), described the computational energy consumed by a CPU for an algorithm A as the Eq. (4.6):

$$E(A) = P_{clk}T(A) + P_wL(A), \tag{4.6}$$

where P_{clk} is a processor clock leakage power, $T(A)$ is the total execution time, $L(A)$ is the total time taken by non-I/O operations, and P_w is used to capture the power consumption per operation performed by the CPU. $T(A)$ and $W(A)$ are estimated using performance features.

Models based on events present some drawbacks, they are highly dependent on the operating system and its architecture, which can complicate portability. There are also limitations regarding the number of simultaneous events that can coexist without adding a non-negligible overhead. Additionally, there are cases where events need multiplexing, for example, when using more hardware events than the CPU can provide. There are also some well-known problems regarding the precision of some events, as shown in many studies Weaver (2008, 2013); Das (2019); Guire (2009); Silva (2019); Silva-de Souza (2020). Some events that should be exact and deterministic (such as the number of executed instructions) show run-to-run variations and over-count on various architectures, even when running in strictly controlled environments. Because of that, our proposed model is not dependent on events and, therefore, not vulnerable to those drawbacks.

Instruction-level energy models, like the one proposed by Yakun et al. Shao and Brooks (2013), estimate energy consumption based on an energy per instruction (EPI) characterization made on Xeon Phi. Their model is expressed as:

$$E(f) = \frac{(p_1 - p_0)(c_1 - c_0)/f}{N}, \tag{4.7}$$

where N is the total number of dynamic instructions, p_0 is the initial idle power, p_1 is the average dynamic power, and $(c_1 - c_0)$ refers to the cumulative number of cycles the micro-benchmark performs and f is the clock frequency. This model is suitable for post-execution analysis when it is possible to count the total cycles. However, it is challenging to use for optimization or forecasting since it does not have an application model

to predict the cycles. For this reason, we propose a model that integrates the behavior of the application taking into account the execution time.

Comprehensive system-level models, such as those by Lewis et al. Lewis (2008) and Mills et al. Mills (2014), consider the energy consumed by various system components.

Lewis et al. Lewis (2008) described the overall system energy consumption using the following equation:

$$E = A_0(E_{proc} + E_{mem}) + A_1 E_{em} + A_2 E_{board} + A_3 E_{hdd}, \tag{4.8}$$

where A_0, A_1, A_2, and A_3 are unknown constants calculated via linear regression analysis and remain constant for a specific server architecture. This model, like the previous one, relies on knowledge of energy spent on each component, making it suitable for post-analysis estimation after the application has already run. However, it is not designed to optimize energy consumption during the actual execution of the application, which is the primary aim of our model.

In another energy consumption model, based on system utilization, Mills et al. Mills (2014) modeled the energy consumed by a compute node with CPU (single) executing at speed σ as Eq. (4.9),

$$E(\sigma, [t_1, t_2]) = \int_{t_1}^{t_2} \sigma^3 + \rho {\sigma_{max}}^3 dt, \tag{4.9}$$

where ρ stands for the overhead power consumed regardless of the processor speed, t_1 and t_2 are the application's initial and final execution times. The overhead includes power consumption by all other system components, such as memory, network, and more. For this reason, although the authors mentioned the energy consumption of a socket, their power model is generalized to the entire server. This model lacks a closed form, i.e., it depends on the definition of $\rho(t)$ to be complete. Our model has a closed form which facilitates analyses.

In reviewing various models related to energy consumption and performance, it is clear that each approach has its unique advantages and limitations. Event-based models, while effective in some contexts, face challenges related to portability and precision. Instruction-level models provide detailed insights but are more suitable for post-execution analysis rather than real-time optimization. Comprehensive system-level

models offer a holistic view of energy consumption but often require extensive component-specific knowledge, limiting their flexibility in dynamic environments.

Overall, these models contribute valuable perspectives to the understanding of energy and performance dynamics in computing systems. However, they also highlight the need for more integrated and application-aware models that can adapt to varying workloads and system configurations. Our proposed model aims to address these gaps by incorporating the behavior of applications into energy and performance predictions, facilitating more accurate and actionable insights for optimization.

Table 4.1 summarizes the existing models comparing the system dependencies and the controllable variables.

Table 4.1: Summary of Related Work on Power Models

Model	System Dependency	Variables	Controllable Variables
Merkel et al. Merkel (2006)	Performance counters	Number of activities	-
Roy et al. Roy (2013)	Performance counters	I/O operations, total time	-
Yakun et al. Shao and Brooks (2013)	Number of instructions	Frequency	Frequency
Lewis et al. Lewis (2008)	Energy of subcomponents	Energy of subcomponents	-
Mills et al. Mills (2014)	Power of subcomponents	Total time, frequency	Frequency
Our model	-	Frequency, cores, input size	Frequency, cores, input size

4.4. Proposed Power Model

The power model can be effectively based on frequency models, which simplify the complex dynamics of processor behavior by focusing on the fundamental components, transistors Rauber (2014); Goel (2016); Du (2017); Gonzalez (1997). This approach extends the behavior of logic gates to the entire architecture, thereby reducing the complexity of power consumption modeling.

The main techniques for manufacturing transistors are FINFET and MOSFET, with FINFET being the more recent and gradually replacing MOSFET. Despite their differences, both technologies can be modeled using common elements of power consumption Rauber (2014); Goel (2016); Du (2017); Gonzalez (1997). These are static power P_{static}, dynamic power P_{dynamic}, and leakage power P_{leak}, which, in combination, comprise and approximate the total power draw.

The dynamic power and leakage power behavior can be approximated by the following equations, respectively, as shown by Sarwar et al. Sarwar (1997) and Butzen et al. Butzen and Ribas (2007).

$$P_{dynamic} = CV^2 f, \tag{4.10}$$

$$P_{leak} \underset{\sim}{\propto} V, \tag{4.11}$$

where C is the load capacitance, V is the voltage applied to the circuit, and f is the switching frequency.

Another common approximation is to assume a linear relationship between the voltage and the applied frequency Usman (2013), such that:

$$f \underset{\sim}{\propto} V, \tag{4.12}$$

These approximations have been demonstrated to be very precise. In the work of Silva et al., the mean percentage error (computed according to Eq. (4.23)) was calculated to be 0.75% Silva (2019).

Thus, the proposed model for one processing core of a multi-core processor is derived by using Equations (10)–(12) to write Eq. (4.13).

$$P(f) = c_1 f^3 + c_2 f + c_3, \tag{4.13}$$

where c_1 c_2, and c_3 are the model's parameters associated with the dynamic, leakage, and static power aspects, respectively. Including the number of active cores p, the proposed estimation of the power consumption of the whole processor becomes Eq. (4.14)

$$P(f, p) = p(c_1 f^3 + c_2 f) + c_3, \tag{4.14}$$

A precise power model can be developed by employing approximations for dynamic and leakage power and assuming a linear relationship between voltage and frequency. This model, extended to account for multiple cores, provides an accurate estimation of the total power consumption of a processor, as demonstrated by its low mean percentage error in empirical studies.

4.5. Propsed Performance Model

To establish a robust performance model for applications, it is crucial to understand the application's behavior in terms of its execution dynamics. This involves modeling the execution time based on the number of instructions, the mean frequency of execution, and the instructions per cycle. Additionally, incorporating the number of processing cores and the parallelism of tasks using Amdahl's law further refines the model. By integrating these factors, along with the input size representing the workload, we can derive a comprehensive equation to predict the execution time of a program under varying conditions.

To establish a performance model, one must first examine the application's behavior. We consider a program a set of instructions executed on a mean frequency f with c_k instructions per cycle to model the application execution time. The time T_f that this program will take to complete at a given frequency is devised as follows:

$$T_f = \frac{I}{c_k f}, \tag{4.15}$$

where I is the total number of instructions and c_k is the ratio of instructions per unit of time.

This model does not yet consider parallel and sequential executions. Thus, the next step is to include the number of cores in the equation. Amdahl's law Amdahl (1967), gives the theoretical background for that. It describes the speedup in latency of the execution of a task at a fixed workload. S, the theoretical speedup of the execution of the whole task, can then be expressed by Eq. (4.16):

$$S = \frac{T_s}{T_p} = \frac{1}{1 - w + \frac{w}{p}}, \tag{4.16}$$

where T_s is the serial time, T_p is the parallel time, w is the proportion of the execution time that benefits from improving system resources, and p is the speedup part of the task that benefits from improved system resources. Combining this with Eq. (4.15), the parallel time at frequency f can be written as:

$$T_p = \frac{T_s}{S} = \frac{T_f}{\frac{1}{1-w+\frac{w}{p}}}, \tag{4.17}$$

We can then write the equation of the program execution time as a function of frequency, the number of cores, and parallelism as Eq. (4.18), and subsequently derive Eq. (4.19):

$$T(f,p) = \frac{I}{\frac{c_k f}{1-w+\frac{w}{p}}}, \tag{4.18}$$

$$T(f,p) = \frac{d_1(p-wp+w)}{fp}, \tag{4.19}$$

where d_1 is a constant.

To fully characterize the application, a parameter called input size N can be introduced to represent the application workload, i.e. the number of basic operations required to complete a problem Kumar (1994). In Oliveira et al. Oliveira (2018), they showed that this parameter could generally be described as exponential. Therefore the proposed performance model is presented in Eq. (4.20). This resulting equation describes the behavior of the execution time of a program for an input size N, frequency f, and p active cores:

$$T(f,p,N) = \frac{d_1 N^{d_2}(p-wp+w)}{fp}, \tag{4.20}$$

where d_1, d_2 and w are constants that depend on the application.

The proposed performance model incorporates key factors such as execution frequency, number of instructions, instructions per cycle, degree of parallelism, and input size. This comprehensive approach allows for a more complete characterization of application performance, facilitating optimization and resource management in IT systems.

4.6. Proposed Energy model

The energy model, discussed in this section, represents a crucial integration of both the power and performance models detailed in previous sections (Sections 4.4 and 4.5).

Combining the power model output described in Section 4.4 and the characterization of the application performance described in Section 4.5,

the total energy can be modeled as:

$$E(f, p, N) = P(f, p) \times \mathrm{T}(f, p, N), \tag{4.21}$$

where $P(f, p)$ is the total power modeled by Eq. (4.14), $T(f, p, N)$ is the execution time estimated by the Eq. (4.20), f is the frequency, p is the number of active cores, and N is the input size. The final equation can be written as:

$$E(f, p, N) = \frac{d_1 N^{d_2}(p - wp + w)(p(c_1 f^3 + c_2 f) + c_3)}{fp}. \tag{4.22}$$

By integrating power and performance models, the equation provides a detailed understanding of energy requirements under different operating conditions. This comprehensive model facilitates informed decision-making regarding resource allocation, optimization strategies, and energy-efficient computing practices.

4.7. Verifying Hypothesis

In this section, we validate whether the assumptions of our model are valid for the system used.

4.7.1 Frequency and Voltage Relation

One of the assumptions was that the frequency and the voltage have a linear relationship, as indicated by Eq. (4.12). To verify that, we build an experiment that sets the frequency to a specific value while sampling the voltage using the APERF and MPERF registers that provide feedback on the current CPU frequency. The average result of the sampling voltages is shown in Fig. 4.1, where we can observe a near-perfect linear relation. This is because manufacturers precisely define those values in the circuit to better suit their design.

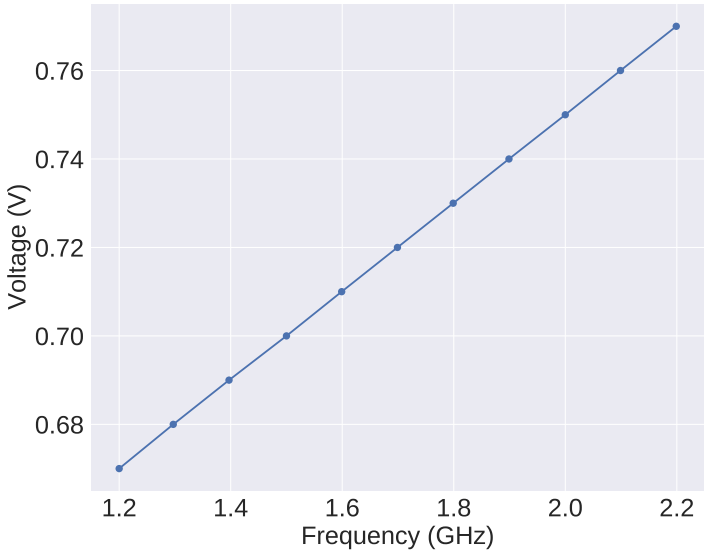
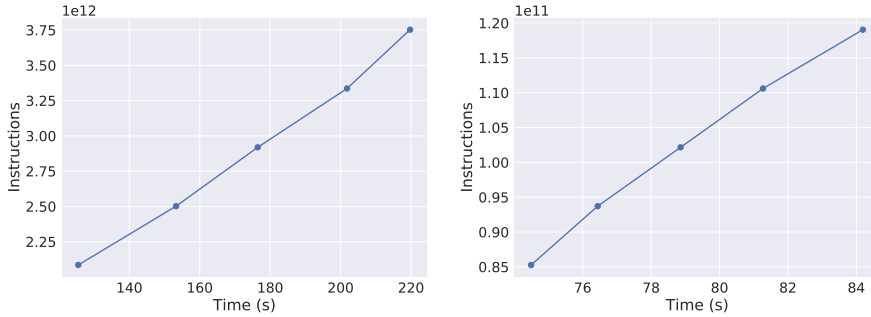


Figure 4.1: Frequency voltage relation.

4.7.2 Input Size and Instructions

We ran the benchmark applications from PARSEC (Section 2.5) with different input workloads assuming linear growth in the amount of work from one input to the other when building our model.

Measuring and controlling the workload would necessitate extensive instrumentation and tuning to determine an input corresponding to a specific workload level. Therefore, assuming that workload is proportional to execution time, we utilize time as a reference for the workload. Figure 4.2 illustrates the validation of this assumption. Table 4.2 shows that the assumption was reasonable since the average correlation was 0.96 for all applications, indicating that growth in the number of instructions will follow the time. This was the case for all applications that we ran in our benchmark and should hold for any data parallelism type of application.



(a) Blackscholes. (b) Canneal.

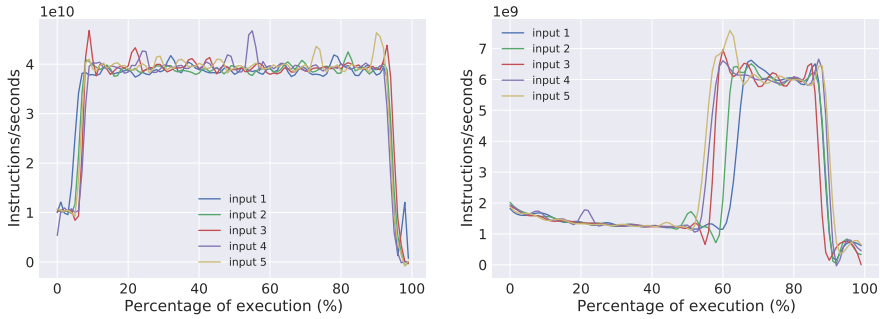
Figure 4.2: Relation between time and instructions for each input size.

The next assumption was that the application's behavior was the same when varying the workload. This condition is necessary for using the model with an unknown input size because, if the behavior is the same, we can interpolate the known inputs. One way to verify this is to measure the rate of instructions per second normalized by the frequency, as shown in Fig. 4.3.

Figure 4.3 shows that between certain workload limits, applications have approximately the same curve when normalized; This happens for all applications in our benchmark.

Table 4.2: Correlation of time and instructions for all applications.

Application	**Correlation**
Blackscholes	0.99
Bodytrack	0.99
Canneal	0.99
Dedup	0.99
Ferret	0.96
Fluidanimate	0.99
Freqmineq	0.99
Openmc	0.94
Raytrace	0.99
Swaptions	0.99
Vips	0.98
x264	0.99
HPL	0.79



(a) Blacksholes. (b) Canneal.

Figure 4.3: Rate of instructions per second varying the input size normalized by the frequency.

The final assumption is that the workload should also not vary depending on the number of cores or frequency. To verify, we measure the total number of executed instructions while varying the cores from 1 to 32. Table 4.3 shows the results.

Table 4.3: Variation of the number of instructions when changing the number of cores for the same input.

Application	**Average Number of Instructions**	**Standard Deviation**	**Standard Deviation (%)**
Vip	7.97×10^{11}	7.16×10^{6}	0.00
Openmc	8.17×10^{7}	1.65×10^{4}	0.02
Rtview	9.91×10^{12}	1.55×10^{9}	0.02
X264	4.52×10^{11}	5.81×10^{7}	0.01
Bodytrack	1.86×10^{12}	3.95×10^{10}	2.13
Fluidanimate	2.09×10^{12}	8.44×10^{10}	4.04
HPL	1.14×10^{8}	1.24×10^{5}	0.11
Blackschole	3.75×10^{12}	1.40×10^{9}	0.04
Dedup	1.02×10^{11}	5.74×10^{7}	0.06
Swapti	2.43×10^{12}	8.87×10^{8}	0.04
Canneal	1.19×10^{11}	4.46×10^{7}	0.04
Freqmine	1.27×10^{12}	4.78×10^{8}	0.04
Ferret	4.76×10^{11}	7.04×10^{7}	0.01

Table 4.3 shows the standard deviation and what that corresponds to in terms of the total number of instructions as a percentage.

The same test was performed for the frequency, varying from 1.2 to 2.2 GHz with 100 MHz steps. The results are shown in Table 4.4.

The examination of various aspects crucial to our model's assumptions reveals a high degree of validity. From the linear relationship between frequency and voltage to the correlation between workload and execution time, each validation underscores the reliability of our model's foundations. With these verifications in place, we can confidently proceed to validate the predictions of our model.

It is also important to notice that when we reference the total number of instructions we are not considering the individual complexity of each one.

Table 4.4: Variation of the number of instructions when changing the frequency for the same input.

Application	**Average Number of Instructions**	**Standard Deviation**	**Standard Deviation (%)**
Vip	7.97×10^{11}	1.16×10^{6}	0.00
Openmc	8.17×10^{7}	4.52×10^{3}	0.01
Rtview	9.91×10^{12}	6.64×10^{5}	0.00
X264	4.52×10^{11}	1.54×10^{5}	0.00
Bodytrack	1.84×10^{12}	2.54×10^{5}	0.00
Fluidanimate	2.38×10^{12}	1.70×10^{9}	0.07
HPL	1.14×10^{8}	5.95×10^{3}	0.01
Blackschole	3.75×10^{12}	4.36×10^{5}	0.00
Dedup	1.02×10^{11}	8.32×10^{7}	0.08
Swapti	2.43×10^{12}	1.48×10^{5}	0.00
Canneal	1.19×10^{11}	3.01×10^{5}	0.00
Freqmine	1.27×10^{12}	3.70×10^{8}	0.03
Ferret	4.76×10^{11}	5.63×10^{7}	0.01

The need for a balance between accuracy and computational feasibility justifies using an average instruction complexity model rather than a detailed instruction-level approach such as Shao and Brooks (2013). While a detailed model might offer higher precision, the average instruction complexity model provides a practical and efficient solution for predicting execution time in diverse scenarios. This approach aligns to develop a performance model that is both informative and manageable, making it suitable for use in real-world applications.

4.8. Fitting the Models

To find the parameters of Eq. (4.22), 10 uniformly random configurations of frequencies (f), cores (p) and inputs (N) were chosen from the range $1 <= p <= 32$, $1.2 <= f <= 2.2$ and $1 <= N <= 5$, respectively. The application was executed for each chosen configuration, and the measured energy and time values were collected. For the input size, if we assume that all CPU instructions take approximately the same time to execute, the number of basic operations will be directly correlated with the time. Thus, we can estimate the input size by looking at the execution time, allowing us to divide a large input size into several smaller ones, knowing their relationship, as performed in the work of Oliveira Oliveira (2018). The unity can also vary depending on the definition. For simplicity, we assign numbers from 1 to 10, increasing the problem linearly, so it is also possible to interpolate any input in between these values.

We aim to fit our energy consumption model to empirical data to ensure accurate predictions of energy usage across various configurations. To achieve this, we executed applications with uniformly random configurations of frequency, number of cores, and input sizes within specified ranges, collecting energy and time measurements. Using uniformly random configurations helps ensure that the model is tested across a diverse range of scenarios, improving its generalizability and reliability.

To find the parameters of Eq. (4.22), 10 uniformly random configurations of frequencies (f), cores (p) and inputs (N) from the ranges $1 <= p <= 32$, $1.2 <= f <= 2.2$ and $1 <= N <= 5$, respectively. We executed the selected application for each chosen configuration and collected the measured energy and time values. Assuming all CPU instructions take

approximately the same time to execute, the number of basic operations can be directly correlated with the execution time. Thus, knowing that relationship, we can estimate the input size by examining the execution time, allowing us to divide a large input size into several smaller ones, as shown in the work of Oliveira Oliveira (2018). The unit of input size can vary depending on the definition, but for simplicity, we assign numbers from 1 to 10, increasing the problem size linearly, and making it possible to interpolate any input size between these values.

For each configuration, power samples were collected using IPMI every second. This sampling rate was chosen based on the magnitude of the mean run time of the applications, which is on the order of minutes. This rate therefore provides enough samples to measure the average power. Additionally, timestamps and total execution time were collected. The total energy spent on each configuration is estimated by first interpolating the power samples using the first-order method and then integrating this function over time.

The model parameters are determined by solving an optimization problem using the nonlinear least-squares method, which minimizes the sum of the squared differences between the predicted and measured values. This approach ensures that the model accurately fits the observed data. Furthermore, we applied machine learning techniques, specifically the Support Vector Regression (SVR) model, to refine our predictions and enhance the model's accuracy. The Python library Scikit-Learn was used to build the SVR model Pedregosa (2011). The SVR was trained using the same data used for parameter estimation of Eq. (4.22) with a grid search used to find the best kernel function and the best values for the hyper-parameters penalty for the wrong (C) and (γ). For this data, the best function was the radial base function (RBF), and the hyper-parameters were $C = 10^4$ and $\gamma = 0.5$.

Through systematic data collection, optimization, and advanced machine learning techniques, our fitting process has yielded a robust energy consumption model, offering reliable predictions across diverse configurations.

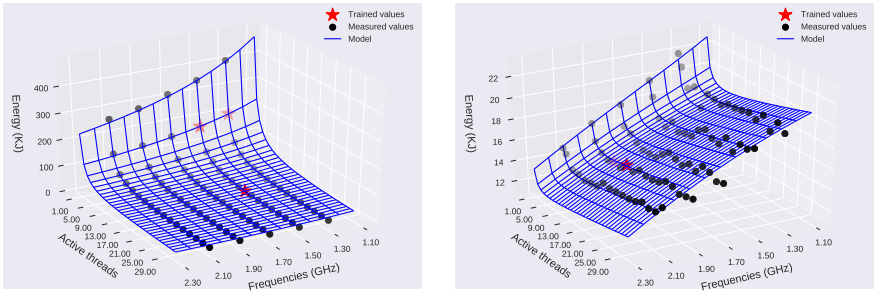
4.9. Measured Versus Modeled Energy

To validate the model, we ran all possible configurations in the tested machine, varying the cores in a range of $1 <= p <= 32$, the frequency in $1.2 <= f <= 2.2$, and the input in $1 <= N <= 5$. The total number of configurations varies from 400 to over 1000 depending on the application, as some applications have restrictions on the number of cores that they can run. Once the data was collected, we computed the mean percentage error (MPE) according to the following equation:

$$MPE = \frac{1}{N} \sum_{i}^{N} \frac{|y_{\text{estimated}} - y_{\text{measured}}|}{y_{\text{measured}}}. \tag{4.23}$$

4.9.1 Frequency X Cores

Figure 4.4 plots the measured and modeled energy consumption for some of the applications modeled. In addition, some of the possible shapes that the model can take while varying the number of active cores, and operating frequency, are shown.

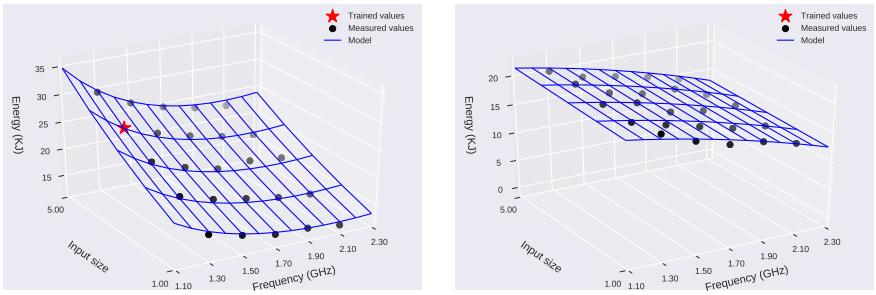


(a) Blackscholes (b) Canneal

Figure 4.4: Example fit for a specific input size. "measured values" are the sensor data, and "minimum energy" is the minimum energy model prediction.

4.9.2 Frequency X Input

Figure 4.5 plots the measured and modeled energy consumption for some of the applications modeled. The diagrams show some of the possible shapes that the model can take while varying the operating frequency and input size.



(a) Blackscholes (b) Canneal

Figure 4.5: Example fit for a specific input size. “measured values” are the sensor data and “minimum energy” is the minimum energy model prediction.

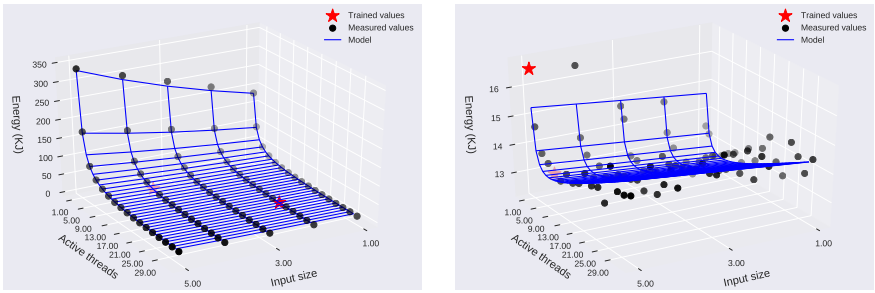
4.9.3 Cores X Input

Figure 4.6 plots the measured and modeled energy consumption for some of the applications modeled. The diagrams show some of the possible shapes that the model can take while varying the number of active cores, and input size.

4.10. Comparison

In this section, the models presented in Sections 4.4 and 4.5 were validated with a benchmark specific for multi-core architectures.

To assess the modeling overhead and accuracy, our proposal was first compared to machine learning approaches. We compared against support vector regression (SVR) Smola (2004), decision tree Kitts (2006), k-nearest



(a) Blackscholes (b) Canneal

Figure 4.6: Example fit for a specific input size. "measured values" are the sensor data and "minimum energy" is the minimum energy model prediction.

neighbors Altman (1992), multilayer perceptron Murtagh (1991), and some new methods, such as Gao et al. Gao (2019).

The complexity of these models can vary significantly. To optimize each model and extract its maximum potential, we employed several mechanisms:

1. **Hyperparameter Tuning:** Adjusting hyperparameters such as the number of hidden layers, the number of neurons in each layer, activation functions, learning rate, and batch size can significantly impact model performance. Techniques such as grid search and random search can be used to find optimal hyperparameter settings.

2. **Regularization:** To prevent overfitting, apply regularization techniques such as L2 regularization (weight decay) and dropout. Dropout randomly sets a fraction of the neurons to zero during training, which helps in reducing overfitting and improving generalization.

3. **Early Stopping:** Monitor the validation performance during training and stop the process when performance no longer improves. This prevents overfitting by stopping training before the model starts to memorize the training data.

4. **Batch Normalization:** Normalize the inputs of each layer to have a mean of zero and a variance of one. This technique can accelerate

training and improve model stability by reducing internal covariate shift.

5. **Learning Rate Scheduling:** Adjust the learning rate dynamically during training. Techniques such as learning rate decay, step decay, or adaptive learning rate methods (e.g., Adam Kingma (2014), RMSprop Tielema (n2012)) can help in achieving better convergence.

6. **Feature Engineering:** Enhance the quality of the input features by creating new features or selecting relevant ones. Feature engineering can significantly impact the performance of the MLP by providing more informative data for learning.

7. **Optimization Algorithms:** Experiment with different optimization algorithms and their variants. Algorithms like Adam, RMSprop, and SGD Bottou (2010) with momentum can offer improved convergence rates and better handling of various optimization landscapes.

Table 4.5 better details the configuration applied on each approach.

Model	**Type**	**Configuration**
Equation	Regression Least Squares Optmizer	Proposed equation model
MLP	Neural Network	Normalization Solver LBFGS Tang (2018) Grid search hyperparameters Activation: logistic, tanh, relu Hidden layers sizes: 100, 300, 500
SVR	Support Vector Regressor	Normalization RBF Kernel Drucker (1997) Grid search hyperparameters
Decision Tree	Decision Tree Regressor	Default settings (SKLEARN (2013))
KNN	K-Nearest Neighbors	Default settings (SKLEARN (2013))

Table 4.5: Summary of Models and Configurations

We computed the mean squared error MSE for all benchmark applications of our study case, as shown in Fig. 4.7. Note that the lowest MSE was achieved by our proposed model. However, SVR was chosen as the most representative of machine learning approaches because it performed best in our tests after fine-tuning.

The average results for each application were calculated using a model

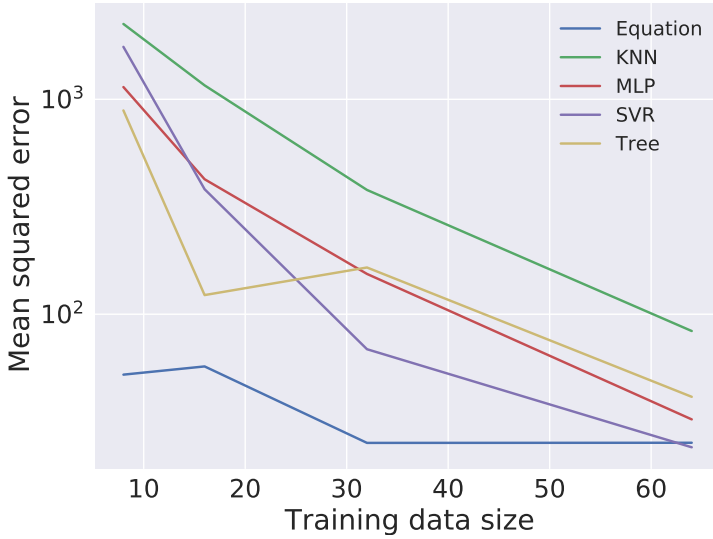


Figure 4.7: Average of the mean squared error for all applications of our study case Section 2.5.

trained with only 10 configurations, and the comparison is displayed Fig. 4.8.

Fig. 4.8 shows that the proposed model always performed better, with a lower MPE than SVR, when we were limited to 10 training points. This result is further explored in the next Section 4.10.1, where we undertake a comparison with different training sizes. The exact values are shown in Table 4.6.

When looking at the MPE results comparing the proposed model and SVR presented in Table 4.6, several key observations emerge. The proposed model always achieves lower MPE values, with the best-case scenario being an MPE of 2.18 for Blackscholes and 2.44 for Freqmine. These represent instances where the model's predictions closely align with the measured energy consumption. Conversely, SVR exhibits higher MPE values, with the worst-case scenario being an MPE of 34.12 for Bodytrack. This indicates significant discrepancies between SVR predictions and actual energy consumption for certain applications. The average MPE for the proposed model across all applications is notably lower compared

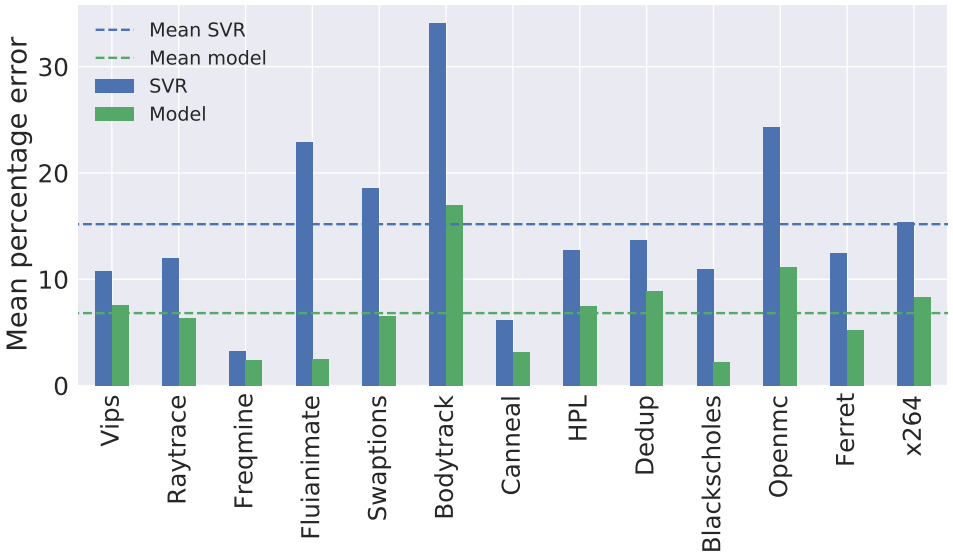


Figure 4.8: Comparison of the mean percentage error between the proposed model and SVR. "Model mean" and "SVR mean" are the average of all MPE values for all applications.

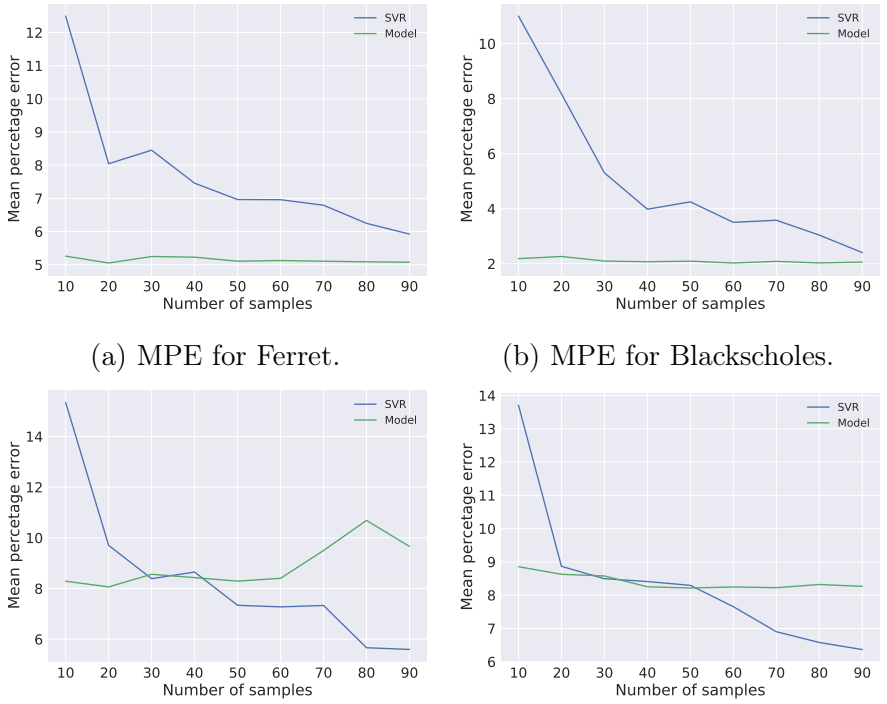
Application	Model	SVR
Ferret	5.25	12.49
Raytrace	6.36	11.95
Fluianimate	2.44	22.90
x264	8.28	15.33
Vips	7.54	10.80
Swaptions	6.54	18.57
Canneal	3.12	6.13
Dedup	8.85	13.70
Freqmine	2.44	3.24
Blackscholes	2.18	11.00
HPL	7.47	12.75
Bodytrack	16.98	34.12
Openmc	11.15	24.34

Table 4.6: Comparison of the Mean Percentage Error between the proposed model and SVR: raw values.

to SVR, further highlighting its superior predictive accuracy.

4.10.1 Overheads on training

It is known that machine learning is data-driven; in that sense, the SVR model obtained using only 10 configurations could be improved, but what about the analytical model? To answer that question, the proposed model and the SVR were also trained with a varying number of configurations. We then compared the MPE and the amount of energy spent to create each model. This accuracy-energy trade-off is crucial since building models' energy overhead defeats the primary goal of saving power when running applications.



(a) MPE for Ferret.

(b) MPE for Blackscholes.

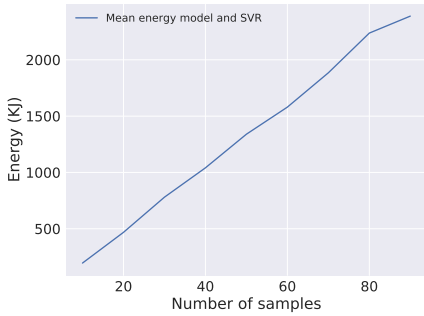
(c) MPE for x264.

(d) MPE for Dedup.

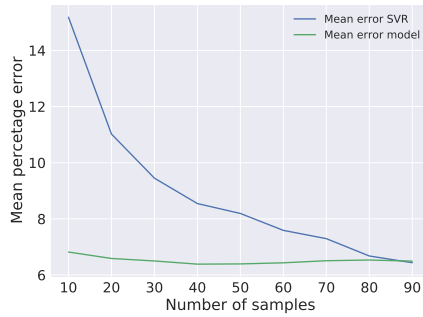
Figure 4.9: MPE of the case studies versus training size, comparing how many training points are necessary to reach an acceptable result.

Fig. 4.9 shows the comparisons of MPE and energy spent to create

each model for two selected applications. According to the results, the analytical model is very stable, not changing much as more data is added, while the SVR keeps reshaping to adapt to the data. The error of the analytical model is almost constant but that of the SVR, initially very high, drops as more data is used in the training process.



(a) Average energy spent on all applications during model creation.



(b) MPE of all applications.

Figure 4.10: Overall results for energy and MPE for each training size.

Figure 4.10 presents the overall results, with the mean energy overhead and MPE for all applications. The meeting point of the MPE for the SVR and the proposed model can be extracted from Fig. 4.10b. It shows that, in around 90 configurations, the SVR starts to have a smaller error. The cost of that is the linear increase in energy spent on training. The increase in energy, about 10 times more, can be observed in Fig. 4.10a.

4.11. Deeper Analysis

One of the most significant advantages of using an analytical model is the understanding of the problem that an equation provides, making many different kinds of analysis possible that are otherwise impossible with a machine learning model. In this section, we discuss one of the possible analyses. In the following figures, we try to understand the contribution of each parameter of the equation to the total energy consumption.

We focused on a specific application and its energy consumption model.

To understand how different settings affect energy usage, we systematically changed one parameter in the model equation while keeping others constant. This allowed us to observe how variations in this parameter influenced the application's energy consumption across different configurations. With the parameter variations in place, we plotted the energy consumption against the application's performance, typically measured in time. Each configuration represented a unique combination of parameter values, resulting in a distinct point on the graph. After that, we computed the Pareto frontier, a set of all Pareto efficient allocations. Each point on the Pareto frontier represents a configuration where no further improvements can be made in one aspect (performance or energy) without sacrificing the other. These configurations offer the best balance between performance and energy efficiency.

By analyzing the application's energy consumption model across various configurations and identifying the Pareto frontier, we pinpointed the configurations where we achieve the most favorable trade-off between performance and energy consumption. This information helps us make informed decisions about resource allocation, ensuring optimal efficiency in our system. Figure 4.11 shows the Pareto frontier for several values for the static power parameter (c_3 in Eq. (4.22)) with configurations of frequency ranging from 1.2 to 5 GHz and cores from 1 to 64, so that we can also have an idea of what is the tendency when we increase the frequency and number of cores.

From this figure, we can see that when increasing the value of the static power parameter, the total energy consumption increases as expected. We can also observe that the values that minimize the total energy consumption on the Y-axis tend to be high frequency and multiple cores. This is one of the consequences of increasing the static power factor. As the dynamic factor proportionally decreases, its variables tend to have less impact on total consumption, enabling configurations with high frequency and several cores. This also enables chip-level optimization for choosing components that change the ratio between static and dynamic power.

Figure 4.12 shows the Pareto frontier in the same ranges described before but for the parameter corresponding to the level of parallelism of the application (w in Eq. (4.22)).

In Figure 4.12, we observe that, as the parallelism level increases the total energy decreases. The number of cores tends to be higher with a

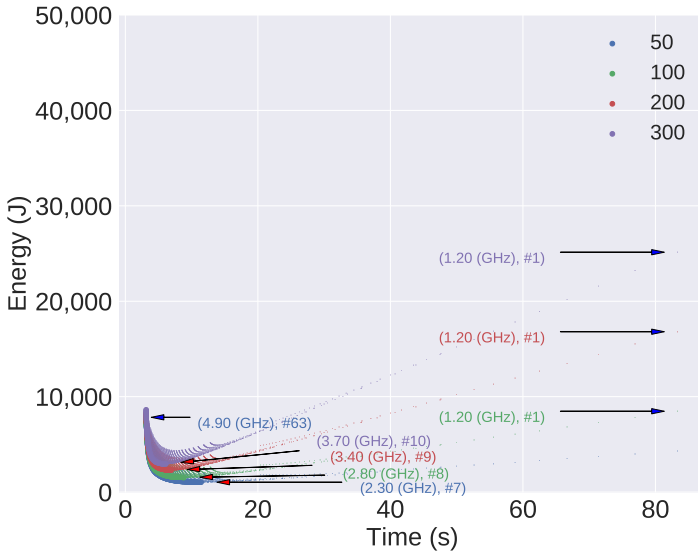


Figure 4.11: Pareto frontier for several values of static power parameter. The arrows with blue heads indicate the maximum energy, while the arrows with redhead the minimal energy for each corresponding curve. The configuration is described by (Frequency, # Number of cores).

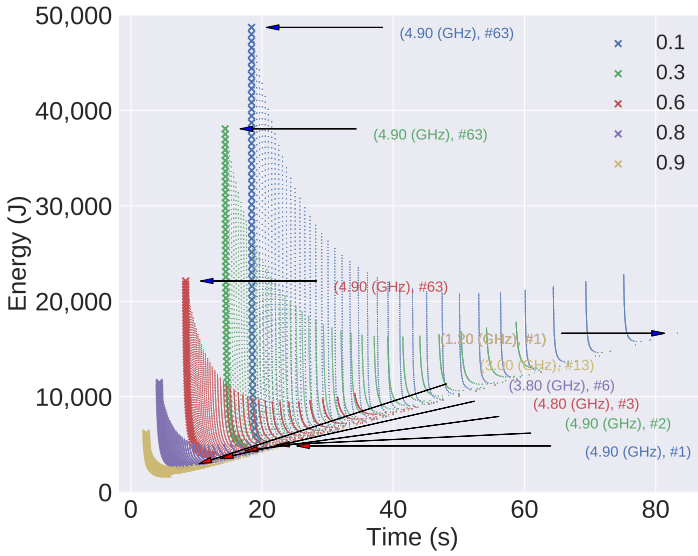


Figure 4.12: Pareto frontier for several levels of parallelism. The arrows with blue heads indicate the maximum energy, while the arrows with redheads, the minimal, for each corresponding curve. The configuration is described by (Frequency, # Number of cores).

higher level of parallelism as expected, and the frequency shows an inverse relation.

4.12. DVFS and DPM optimization

In the realm of power management optimization for HPC systems, effective strategies can lead to substantial energy savings. This section delves into the evaluation of our proposed approach for optimizing DVFS and DPM. By leveraging a simple algorithm to determine the optimal frequency and number of active cores based on a prescribed equation, we aim to surpass the power management choices offered by default in the Linux operating system.

The effectiveness of the proposed approach during optimization was evaluated with a simple algorithm that finds the optimal frequency and number of active cores from the proposed equation. The results were then compared to the Linux default choices for power management.

With Eq. (4.22), it is possible to calculate energy consumption estimates for each possible configuration since there is a finite range of possible values for the frequency and number of cores. It is also possible to apply constraints on the execution time, frequency, and the number of active cores. Then, the configuration that minimizes energy consumption for a given input can be selected. The complete workflow is shown in Fig. 4.13. We can see that any optimization problem can be structured with our model and the system's constraints.

In the following examples, the optimization problem that we build is to minimize the energy equation given the constraints of possible frequencies and the number of cores that our system can run. The algorithm selected to minimize was the newton-CG Royer (2020). Current HPC managers leave to the user the choice of how many cores to use. On this basis, three situations were analyzed concerning the number of cores:

1. Worst choice: number of cores that maximize the total energy consumed;

2. Random choice: energy consumed for a random choice of the number of cores;

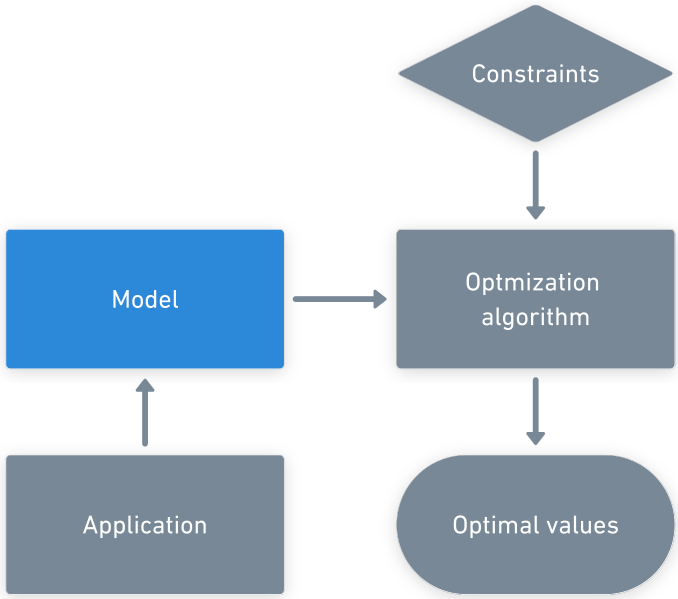


Figure 4.13: Optimization workflow showing how DVFS and DPM optimization could be implemented from ou model.

3. Best choice: number of cores that minimize the total energy consumed (oracle).

The default option for the Linux governor is Ondemand, and, by default, it has no DPM control for the number of active cores. As Ondemand only performs DVFS, for comparison, each application was executed with all available cores in the system, from 1 to 32. Figs. 4.14 and 4.16, show the energy savings for Ondemand, i.e., Eq. (4.24) for the three cases described above. The savings and losses for each case are:

$$\frac{Ondemand - Model_{min}}{Ondemand} \tag{4.24}$$

1. Worst choice: save 69.88% on average;

2. Random choice: save 12.04% on average;

3. Best choice: lost 14.06% on average.

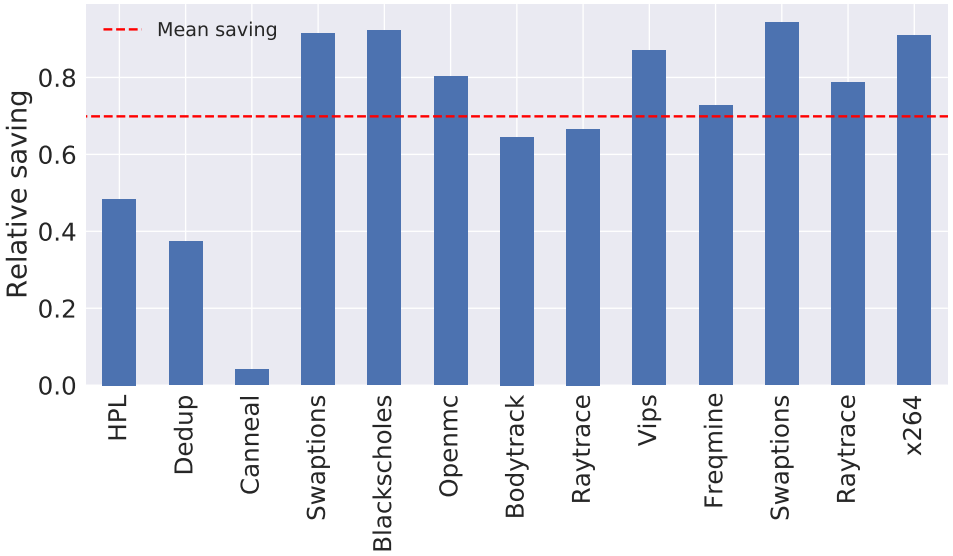


Figure 4.14: Energy savings comparisons between the proposed model and the Worst case.

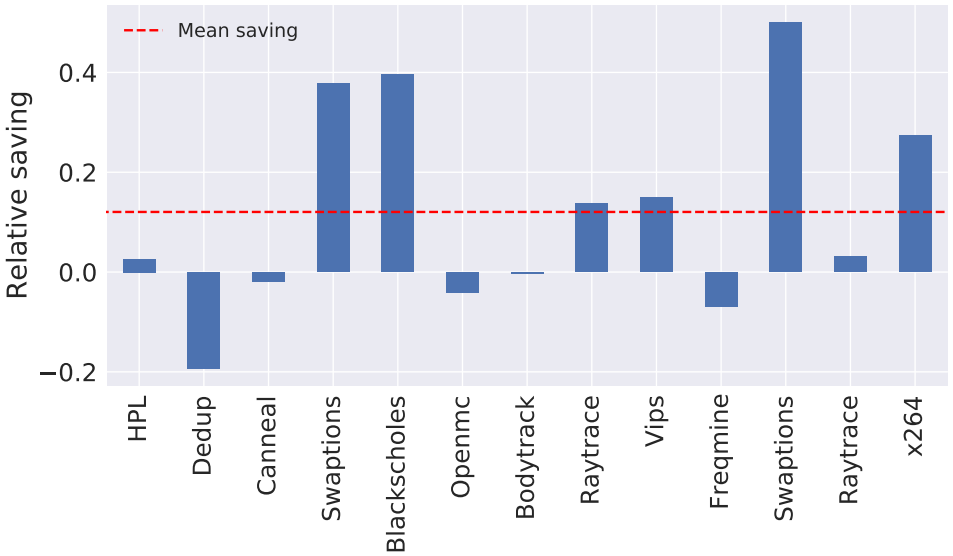


Figure 4.15: Energy savings comparisons between the proposed model and the Random case.

By default, operating systems do not implement DPM at the core level, and, in HPC, the user usually explicitly chooses the number of cores to run their job. To give a better idea of the impact on the energy consumption of DPM at the core level, we analyzed the choices of the number of cores over one year in the HPC center at UFRN. The result is plotted in Fig. 4.17. It is noteworthy that the most common choice among regular users is to request a single core per job, which corresponds to the worst-case scenario for all applications analyzed in this investigation. The best choice was quite often 32 cores, which is the third most popular choice among users, but it is 72 times less frequent than 1 core. This led us to envision how much energy could be saved and encouraged us towards future research using the proposed model for DPM or more advanced optimization algorithms. In practice, this approach can be implemented by allowing the resource manager to perform these changes for the user using pre-scripts and post-scripts for high-energy consumption job submissions.

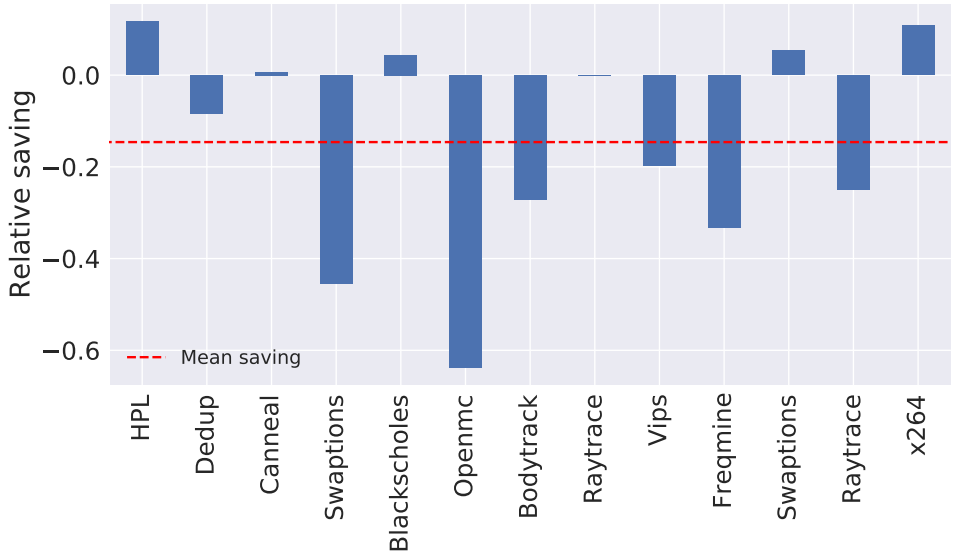


Figure 4.16: Energy savings comparisons between the proposed model and the Best case.

In conclusion, our exploration into DVFS and DPM optimization

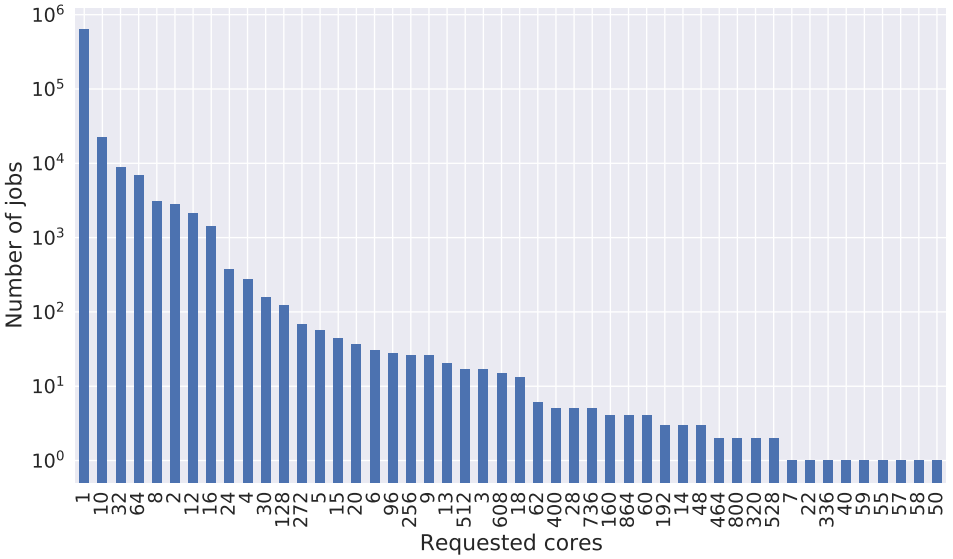


Figure 4.17: Number of CPU requests during one year in HPC cluster, sorted by the number of cores requested per job.

strategies reveals promising avenues for enhancing energy efficiency in HPC environments. Through meticulous analysis and comparison, we've demonstrated the potential of our proposed approach to outperform default power management schemes, offering substantial energy savings. By aligning configurations with the Pareto frontier, we've identified optimal trade-offs between performance and energy consumption, laying the groundwork for future research and implementation in resource management systems. This research underscores the significance of tailored power management strategies in driving sustainability and cost-effectiveness in high-performance computing.

CHAPTER 5

Application-Phase

While substantial research has been conducted on DVFS and DPM, an underexplored aspect remains: the effect of phase division choices on energy consumption.

This chapter delves into how phase division impacts the energy efficiency of algorithms. We introduce a methodology that leverages measurement data combined with a heuristic approach to guide the selection of optimal phase divisions. Our heuristic significantly narrows the search space from 10^{7000} to 10^2, achieving an average error of 10% and reducing energy consumption by up to 38% compared to standard Linux DVFS.

Additionally, we evaluate the trade-offs associated with having numerous phase divisions and the resultant overhead. Our findings indicate a practical limit to the number of beneficial phases for a running application, establishing a lower bound for the minimum number of effective phases.

By exploring these factors, this chapter aims to provide a deeper understanding of phase division's role in optimizing energy consumption in high-performance computing systems.

5.1. The Effect of Phase Division Choices on Energy Consumption

Up to this point, we have been considering the application as a single workload for our modeling efforts. However, in real-world scenarios, the application workload can vary due to many factors, such as changing input sizes, varying computational demands, and different execution paths. Consequently, it is crucial to consider the different phases an application can undergo. Each phase may exhibit unique characteristics and resource requirements, which, if accounted for, can lead to more precise and effective energy optimization strategies. This chapter, therefore, focuses on analyzing and optimizing the energy consumption of applications by taking into account their distinct phases.

Most studies on software energy-saving primarily focus on workload-based optimization algorithms, with fewer addressing the division of applications into distinct phases—segments with specific execution parameters. A common approach is to divide the application into equal time slices, which simplifies optimization and ensures that the algorithms remain lightweight. However, this method can lead to suboptimal energy savings, as different phase divisions might yield better results. Additionally, there is an overhead associated with switching configurations, which could be reduced by determining the optimal timing for these switches. This optimization may require more complex algorithms but can result in more precise energy savings. The key challenges remaining are to determine the optimal number of phases and to pinpoint the exact start and end times for each phase.

Indeed, programs typically progress through multiple execution phases, each characterized by a specific behavior. Thus, considering multiple fixed configurations, one for each phase of the program's execution can offer additional flexibility in reducing energy consumption.

We propose a methodology to analyze the power profiles of applications at different execution conditions to determine the most promising phase divisions, as well as to provide practical boundaries and estimates for energy gains. The main contributions of this work are:

- We leverage a prior-knowledge measurement campaign to estimate

optimal phase divisions in multi-core applications;

- The use of only single-phase measurements in the campaign ensures low overhead;
- The cost function we proposed based on these measurements allows phase-division optimizations;
- For all applications analyzed, we show that the optimal energy does not improve much over a few tens of phases.
- Our experiments show that the optimal phase divisions outperform the Linux default governor.
- On average, our approach uses 38% less energy than the default Linux governor with the optimal, maximal, and minimal amount of active core counts, respectively.

In this chapter, we explore the impact of phase division on energy consumption and propose a methodology to optimize energy usage in high-performance computing applications. We begin by discussing the complexity of the problem in Section 5.2 follow by Section 5.3, examining existing research on phase division and its influence on energy efficiency. Next, we detail the Section 5.4, introducing our approach to identify energy-efficient phase divisions through a prior-knowledge measurement campaign.

Subsequently, we present our Section 5.5, outlining the algorithm for estimating the energy consumption of a single phase, and then extend this to Section 5.6, where we combine multiple phases to estimate overall energy consumption. We then describe the Section 5.7, where we employ optimization techniques to determine the most energy-efficient phase divisions.

In Section 5.8, we showcase the results of our experiments, demonstrating the effectiveness of our proposed methods. Finally, we introduce the concept of Section 5.9, a novel approach to characterize application parameters and models, further enhancing our understanding of phase behavior in applications.

5.2. Prior-knowledge Measurement Campaign

The growing complexity of modern computing applications necessitates efficient methods to optimize energy consumption. To achieve this, we propose an application-phase division algorithm supported by the characterization and modeling of applications on specific CPU targets. This approach helps identify both energy-moderate and energy-intensive phases.

Naively identifying energy-optimal phase divisions using direct power and performance measurements would require evaluating a vast number of possible configurations. This approach is impractical because it would necessitate evaluating each phase of the phase-division solution across all possible power configurations, defined by combinations of frequency and the number of active cores. The operating frequency may range from the minimum processor frequency to the maximum, often in fixed steps. The number of cores is also discrete and has increased significantly with the advent of multi-core processors. Brute force or other sampling methods would be infeasible, even with hardware constraints limiting the discrete values.

For example, let d_t be the minimum time interval a processor remains in a given power configuration, T be the total application time, and C be the number of possible power configurations. The number of possible phase-division solutions can be estimated by:

$$\left(\frac{T}{d_t}\right)^C.$$

. This number grows astronomically because d_t is in the order of microseconds, T is in the order of seconds, minutes, or even hours, and C is in the order of hundreds or thousands, leading to an estimate in the order of 10^{7000}.

To circumvent this problem, we propose a heuristic that balances accuracy with hardware limitations, making the analysis feasible. For a specific application and workload, assuming that the power configuration primarily affects speed or duration and negligible power is consumed during phase switching, we measure the power profiles of single executions for each power configuration. In brief, we run and measure the entire application across all possible machine power settings (frequency-voltage

and number of cores). We then estimate power profiles for any phase-division solutions by consolidating the time intervals of the solution (phases) with the measured power profiles, as described in the next section. This approach drastically reduces the number of necessary runs to explore the search space to the order of hundreds. It allows for swift and fairly accurate power consumption estimates for alternative phase divisions based on real measurements.

5.3. Phase Division Related Work

Previous works in energy optimization of computer systems have primarily focused on techniques such as DVFS and DPM to control hardware resources such as the processor's operating frequency and the number of active processor cores. These techniques have been widely studied and implemented in various systems, including mobile devices and HPC servers.

Scheduling algorithms have also been proposed as a way to optimize energy consumption, managing the execution of tasks with deadlines while taking processor parameters into account. For example, Irani et al. Irani (2007) formalized the problem of scheduling incoming jobs to minimize total energy consumption. Saha et al. Saha (2012) evaluated various Real-Time DVFS (RT-DVFS) schedulers through implementation and measurements. However, most evaluations were based on simulations, drawing attention to the fact that real-time measurements can affect execution, leading to contradictory results.

Feedback-based approaches have also been proposed, as in Poellabauer et al. Poellabauer (2005), where, while running the application, the CPU behavior is used to predict CPU requirements. However, in these approaches, mispredictions can lead to missed deadlines, sub-optimal energy savings, and significant overheads, with frequent changes in the chosen frequency or voltage. One shortcoming of previous approaches is that they don't consider other "indicators" of future CPU requirements, such as frequent I/O operations, memory accesses, or interrupts.

Although most papers consider the deadline, some also take job division as part of the optimization problem. In the paper of Agrawal et al. Agrawal (2021), the authors show that if the jobs can be divided into

arbitrary parts, a minimum-energy schedule can be generated in linear time, giving exact scheduling algorithms. However, they provided proof that the scheduling problem is NP-hard when jobs are not divisible while also giving approximation techniques with boundary constraints.

When it comes to energy savings, it is not just a matter of speeding up execution to meet a deadline. Indeed, choosing a different deadline could result in quite different energy savings. However, if we split an application into phases, by knowing the proper CPU settings for each execution interval, we could optimally manage resources and, consequently, energy. For that, regardless of the workload that would impact execution, identifying the phase segments and their number and correlating these phases to an ideal CPU configuration would be necessary.

While many works consider only the deadline, some also integrate the division of jobs into the optimization problem. Agrawal et al. Agrawal (2021) demonstrate that if the jobs can be divided into arbitrary parts, the generation of a minimum-energy schedule can be achieved in linear time, by proposing exact scheduling algorithms. In the scenarios where jobs are non-divisible, they claimed proof that the scheduling problems are NP-hard, and propose bounded approximation algorithms. However, this approach runs into difficulties as various constraints limit the optimization problem. Our approach, on the other hand, works with measured data, enabling us to find solutions independent of these constraints.

All these works have, at some point, taken into account a deadline (specified by the user or the system) which delimits the phases of the execution. This implies that choosing a different deadline could result in quite different energy savings. In our proposal, we first study the impact on energy consumption of dividing an application into phases. In this context, we estimate the amount of energy that could be saved if an algorithm could somehow give the ideal deadline.

5.4. Phase Division Proposed Approach

In this section, we describe the proposed approach to estimate the energy of a given phase from a phase-division solution in a given power profile. This is achieved using the power profiles collected when running the application with fixed power configurations. It is worth noting that

variations in a power profile are solely related to variations in the instructions executed by the application program, including their impact on the memory hierarchy dynamics, since frequency and number of cores are kept constant.

To find the optimal phase division, we need to ensure that the chosen division is the most energy-efficient among all possible divisions and combinations of power configurations. This mathematical problem can be modeled in various ways, but always with some concessions to make it solvable. A brute-force approach would be impractical due to the infinite possibilities for splitting phases, although some hardware limitations make our analysis more feasible. Hence, we propose a heuristic approach that aims to achieve results as close as possible to brute force, while considering hardware constraints to ensure viability. One primary constraint is the discrete speeds at which the processor can operate.

Continuous division is impractical and unrealistic, limiting our analysis to discrete time intervals corresponding to the processor's maximum performance speed. This constraint significantly reduces our exploration space, yet it remains vast. For example, let d_t be the minimum processor action interval, T be the total application time, and C be the number of power settings. The number of possibilities can be estimated by:

$$\left(\frac{T}{d_t}\right)^C.$$

This number grows astronomically because d_t is in the range of microseconds, T is in the range of seconds or minutes, and C is in the range of hundreds, leading to an estimate in the range of 10^{7000}.

Each configuration provides a distinct power profile for a given application, and phase division represents a combination of these configuration settings. This insight allows us to rationalize the exploration of the solution space. If we can run each power setting once and provide a method to combine the resulting power profiles, our problem becomes more manageable. To make this possible, we assume that configuration changes only affect the program's duration, not its behavior, ensuring that at each division, the energy consumption remains independent of previous configuration settings.

Our approach aims to combine speed and precision in modeling, prioritizing efficiency without compromising accuracy. Consequently, our

algorithm uses real data collected from an actual system. The idea is to first run the application with all possible configurations (e.g., frequency and number of active cores) of the machine without time division, and then estimate the power consumption for different combinations of phase divisions. In summary, our approach aims to identify the optimal phase division to minimize energy consumption for specific applications by leveraging real system data for accurate modeling. Additionally, we ensure that the modeling process remains both swift and precise.

5.5. Energy Estimation Algorithm For a Single Phase

To analyze the impact of configurations and identify the optimal phase division, we propose a zero-order integrator, which calculates the energy consumption in an interval for a given configuration. The integrator uses the data gathering method described in Section 5.8.1, leveraging energy measurements and power profiles for an application running under different power configurations.

The integrator algorithm computes energy consumption by integrating power over time within specified phases of the application's execution. Figure 5.1 illustrates an example of energy computation for an application with four different power configurations. The phases are defined as percentages of the total execution time. Assuming the phases are already defined, the integrator processes each phase based on the selected power profile, providing the total energy consumption as a result.

The algorithm for the integrator is described as follows:

```
1 struct Info {
2   std::vector<double> time; // sample time
3   std::vector<double> power; // sample power
4   double start_time;
5   double stop_time;
6   double elapsed_time;
7 };
8 double integrator(Info& data, double start, double stop)
9 {
```

```
10   auto t1 = data.start_time
11       + data.elapsed_time * start;
12   auto t2 = data.start_time
13       + data.elapsed_time * stop;
14   auto v1 = lower_bound(data.time.begin(),
15               data.time.end(), t1)
16               - data.time.begin();
17   auto v2 = lower_bound(data.time.begin() + v1,
18               data.time.end(), t2)
19               - data.time.begin();
20   if (v1 == v2)
21     return (t2 - t1) * data.power[v1];
22   auto en = (data.time[v1 + 1] - t1) * data.power[v1]
23       + (t2 - data.time[v2]) * data.power[v2];
24
25 #pragma omp simd
26   for (auto i = v1 + 1; i < v2; i++)
27     en += (data.time[i + 1] - data.time[i]) * data.power[i];
28
29   return en;
30 }
```

Figure 5.1 shows the energy estimation of the four phases of an application in four different power profiles.

The proposed algorithm calculates the energy within the specified time range of the phase by linearly interpolating power values between sample points. It takes a reference to an `Info` object representing time-series data, along with the start and stop times for the integration interval. The `Info` struct contains vectors of sample times (`time`) and corresponding power values (`power`), along with metadata about the time range represented by the data. The algorithm first computes the corresponding indices (`v1` and `v2`) of the sample points closest to the start and stop times, respectively, using binary search (`lower_bound`). Then, it calculates the energy within the interval by summing up the contributions from the samples falling within the interval. The contributions are computed based on the linear interpolation of power values between adjacent sample points. If the start and stop times fall within the same sample interval (`v1 == v2`), the energy contribution is calculated using the power value at the closest

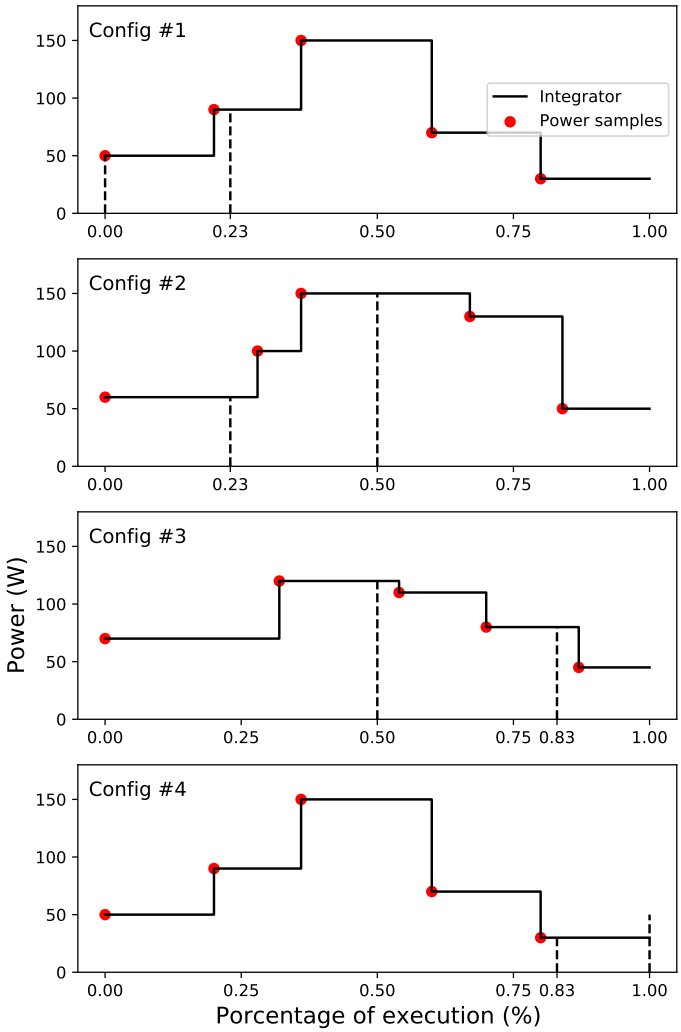


Figure 5.1: Power vs. percentage of execution for a given application in 4 power profiles. For each profile, the red dots represent the power samples, the dashed vertical lines define the start and stop time intervals of the phase, and the hatched area is the estimated phase energy.

sample point.

5.6. Energy Estimation Combining Multiple Phases

To estimate the energy consumption of an arbitrary phase division, we consider minimizing the total energy over all possible combinations of phases and power profiles. Instead, we adopt a per-phase approach that reduces complexity. Given the power profiles obtained during the prior-knowledge measurement campaign and a list of break points corresponding to the phase divisions, we select a power profile for each phase that results in the lowest power consumption for that phase.

The algorithm takes two inputs: a list of configurations and a list of phase divisions for an application. For each phase, it iterates through each configuration and calculates the energy consumption of the phase using the integrator algorithm presented in Section 5.5, which calculates the energy consumption by integrating the power samples over the phase time interval.

For each phase, the algorithm selects the configuration with the lowest energy consumption and adds this minimum energy consumption to the total energy consumption of the application. The output is the total energy consumption of the application for a given set of phases, as shown in the following algorithm.

```
1  double phase_optimization(Pascal& data,
2                vector<double>& phase)
3  {
4    double total_en = 0.0;
5    for (size_t i = 0; i < phase.size() - 1; i++)
6    {
7      auto min_en = numeric_limits<double>::max();
8      for (auto& info : data.infos)
9      {
10       auto en = integrator(info,
11                    phase[i],
12                    phase[i + 1]);
```

```
13        if (en >= 0 && en < min_en)
14          min_en = en;
15      }
16      if (min_en == numeric_limits<double>::max())
17        return -1; // Error
18      total_en += min_en;
19    }
20    return total_en;
21  }
```

This algorithm offers a comprehensive view of the energy consumption of a phase-division solution without requiring additional energy measurements. Given a specified set of phases, it can effectively choose optimal power profiles for each application phase by pinpointing the least energy-consuming power settings. As such, it can be directly employed to schedule the phases aiming for energy savings without compromising performance.

The low-overhead calculation of phase energy and the minimum total energy of a given phase-division solution for an application allows for optimization of the time intervals that define the phases. The next subsection presents an approach for such optimization.

5.7. Optimizing Phase Division

Here, we describe the algorithm employed to optimize the phase divisions, which involves determining the duration and location of each phase. Essentially, the algorithm solves a minimization problem to determine the optimal time boundaries of each phase, given a specific number of phases. To achieve this, we use the algorithm outlined in Section 5.6 as an objective function. The goal is to find the phase divisions that minimize the total energy consumption.

In this approach, a Genetic Algorithm (GA) serves as the solving mechanism. This involves establishing the initial population with a specified granularity, overseeing the mutation process, and using the objective function to steer the evolution towards the desired outcome.

While other search algorithms could be applied to this problem, Ge-

netic Algorithms offer several advantages that make them particularly well-suited for this task. Their ability to handle complex and large search spaces, balance exploration (searching new areas of the search space) and exploitation (refining known good solutions), and provide a robust and proven approach to optimization. The specific choices of population size, number of generations, mutation rate, and crossover mechanisms are tailored to balance exploration and exploitation, ensuring a complete and effective optimization process.

For instance, to study the phases in the range of 3 to 100 divisions, we use the GA with a population of 10^3 individuals. Over 300 generations, we keep the best 10% of individuals from each generation while maintaining a mutation rate of 10%. The reproduction function combines half of the phases of two individuals, and the mutation randomly changes one division. By randomly assigning the initial positions of the phases, a wide range of possibilities are considered early in the optimization process, increasing the chances of finding the optimal solution. This approach ensures that the optimization process is thorough and not limited by initial assumptions about phase size and position.

5.8. Experimental Results

The next two Subsections present the system and applications used in the experiments, respectively. Following we describe how the measurements were made and, in the last tow Subsections, we present the experiments and the their results. The applications and setup for this experiment are described in Section 2.4 and Section 2.5.

5.8.1 Data Gathering

Using the our framework Chapter 3, the data was collected by running applications in all possible power configurations. In the system we used, that is 32 cores and 13 frequencies, which means 416 (32*13) possible power configurations. Power samples were taken from dedicated sensors in the IPMI system with a sample rate of 0.5 seconds.

5.8.2 Experiments with the number of phases

We used the applications described in Section 2.5 to compute the optimal phase selection. Fig. 5.2 indicates, for a set of representative applications, the energy consumption according to the number of phases compared to an optimized single-phase.

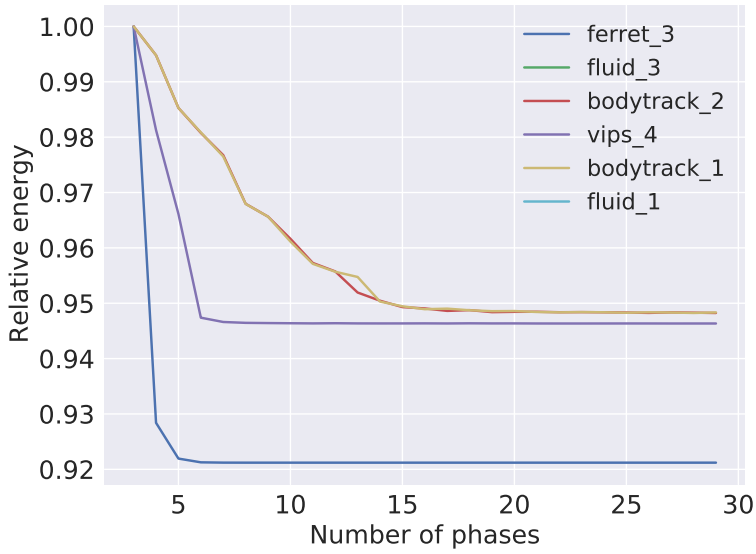


Figure 5.2: Relative energy vs the number of phases using applications from PARSEC 3.0, HPC, and Openmc benchmarks with different sizes' inputs.

Figure 5.2 shows very interesting results. The energy consumption of the optimal single-phase configuration is always higher for all cases. Furthermore, it is not necessary to divide the application into many tens of phases. The amount of energy reduced beyond 35 phases was negligible. In detail, Figure 5.3 illustrates the optimal number of phases for all our tests. We can see that most applications reach the ideal configuration with less than 20 phases, while there is still a uniform distribution to a higher number of phases.

These results meet our expectation that the optimum for n phases should always be better than or equal to a single phase. This becomes

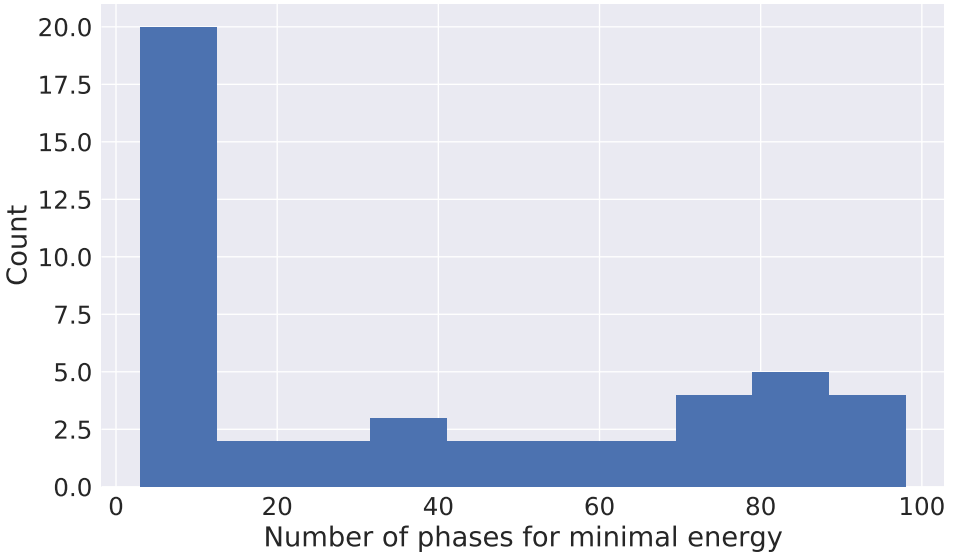
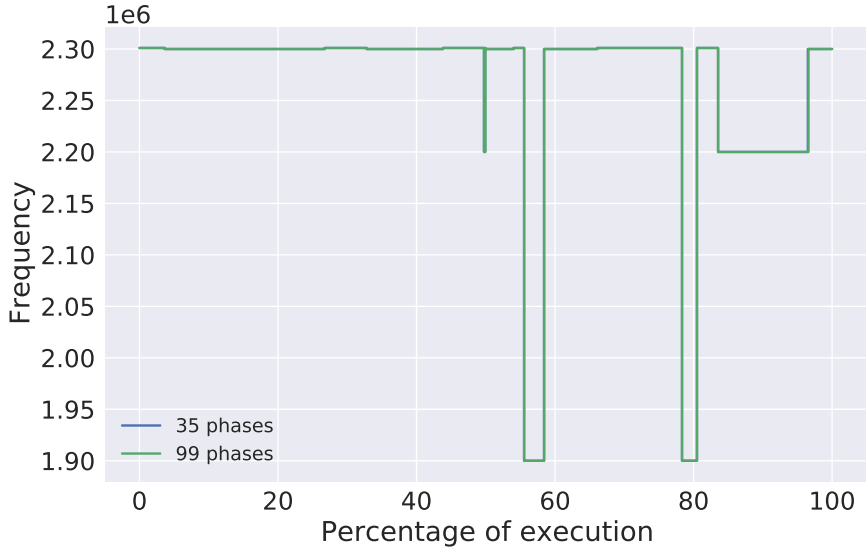


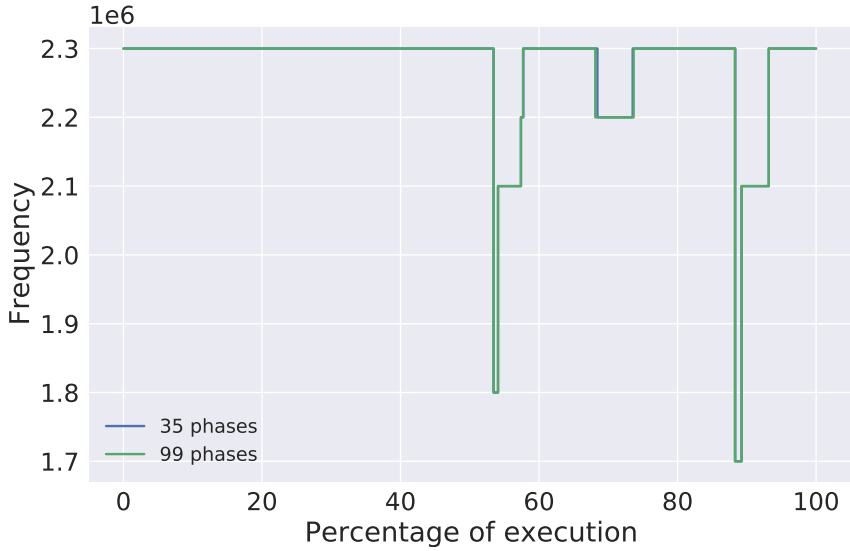
Figure 5.3: Histogram showing the frequency of the optimal number of phases for all applications

clearer when we look at the number of cores in Fig. 5.5a and Fig. 5.5b, and frequencies in Fig. 5.4a and Fig. 5.4b for two particular applications, Bodytrack and Black-Scholes. These applications demonstrate the optimal frequency and cores do not fundamentally change between 35 and 99 phases. Moreover, the result shows that the phase granularity does not fundamentally change the locations of the phases. This result is particularly important when characterizing applications and identifying behaviors independent of the hardware resources used.

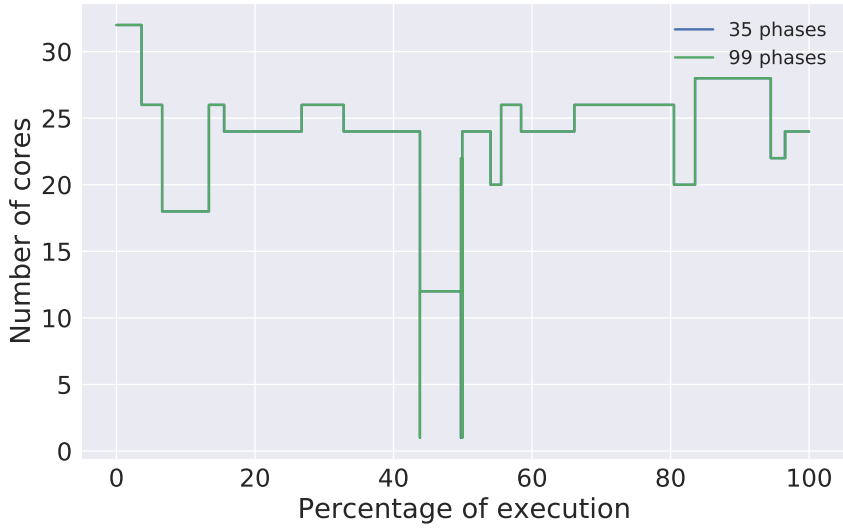
When examining the behavior of a particular application, we can observe how energy consumption varies with an increasing number of phases. Comparing Fig. 5.6a and Fig. 5.6b, we notice a more detailed breakdown of energy consumption with a higher phase count. This aids in pinpointing energy-intensive phases, identifying inefficiencies in code sections, and assessing hardware utilization. It also sheds light on scalability attributes with varying input sizes. Nonetheless, it's important to note that a higher number of phases may introduce added complexity and analysis overhead.



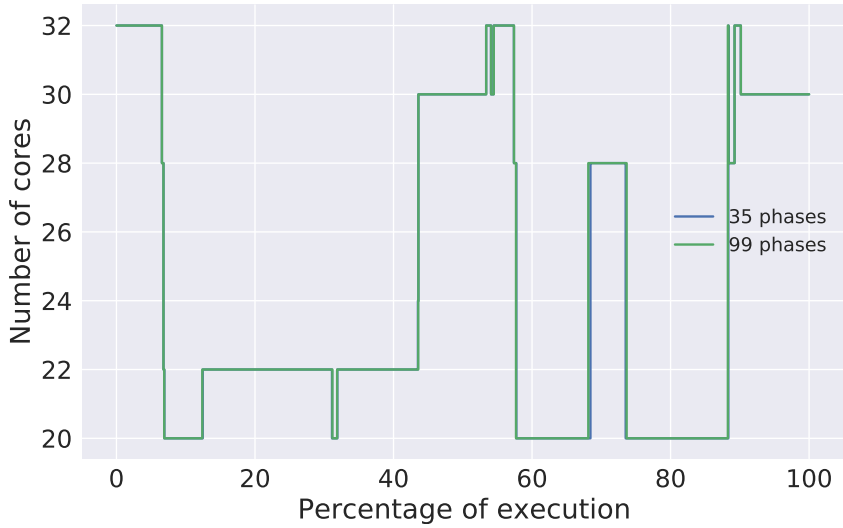
(a) Execution frequency vs. percentage of execution for the Bodytrack application, comparison when using 35 and 99 phases.



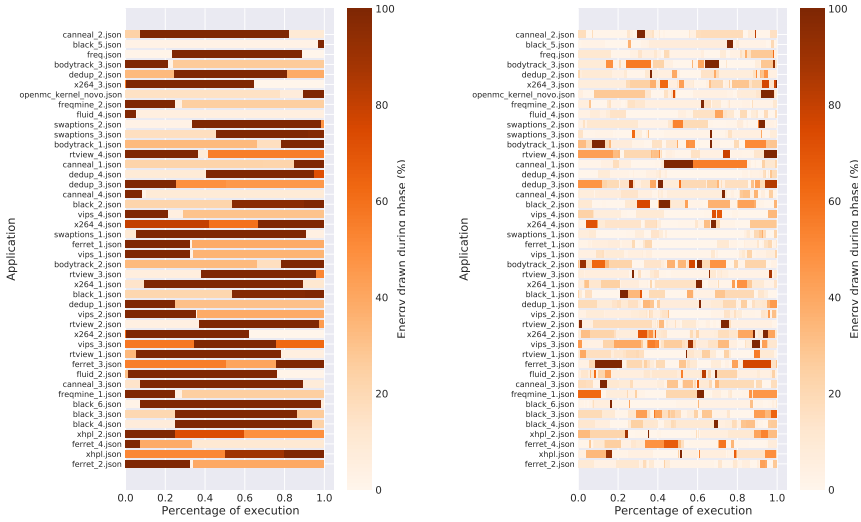
(b) Execution frequency vs. percentage of execution for the Black-Scholes application, comparison when using 35 and 99 phases.



(a) Number of cores vs. percentage of execution for the Bodytrack application, comparison when using 35 and 99 phases.



(b) Number of cores vs. percentage of execution for the Black-Scholes application, comparison when using 35 and 99 phases.



(a) 3 divisions.

(b) 35 divisions.

Figure 5.6: Phase division heatmaps showing the energy consumption per phase for all applications with different numbers of phases.

5.8.3 Comparing against the default Linux governor

Using the same metrics of Section 4.12, we analyze this method on the worst, random, and best choices.

When we compare this method with the default DVFS algorithm in Linux, we observe an average saving of 38%, as shown in Figures 5.7, 5.8 and 5.9, the relative energy per application with different input sizes.

5.9. Application Fingerprint

In our approach, applications are characterized using metrics collected during their execution by performance counters. We can extract detailed information from performance counters, such as CPU and memory usage, disk access, or network traffic. For future work, it is important to reduce further the number of data variants collected, to provide rapid identification of phase locations for a particular given granularity, and to use this information to provide a characterization of the program that can easily

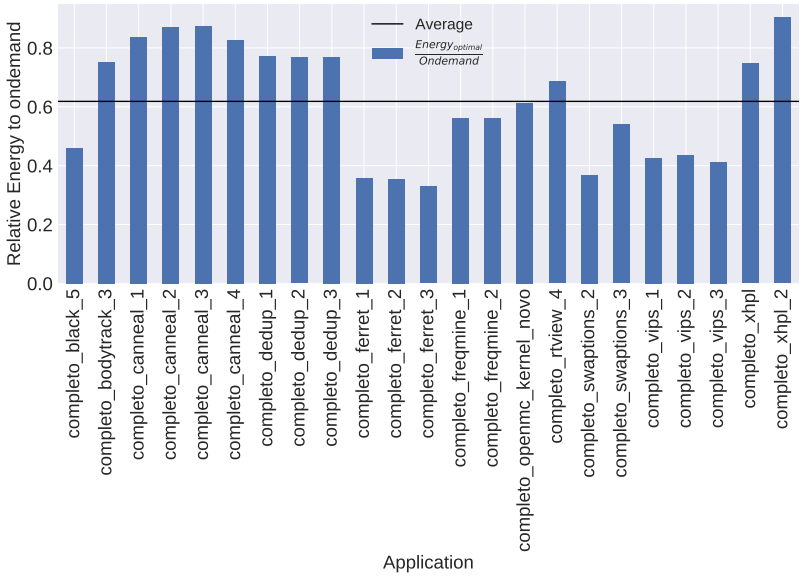


Figure 5.7: Optimal phase splitting energy vs. on-demand governor on Linux: relative energy comparison for applications with different input sizes for the worst choice case. Lower is better.

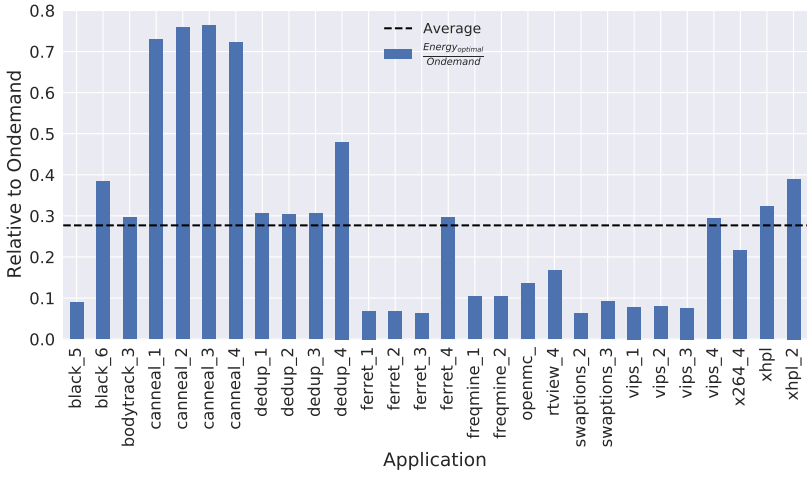


Figure 5.8: Optimal phase splitting energy vs. on-demand governor on Linux: relative energy comparison for applications with different input sizes for the random choice case. Lower is better.

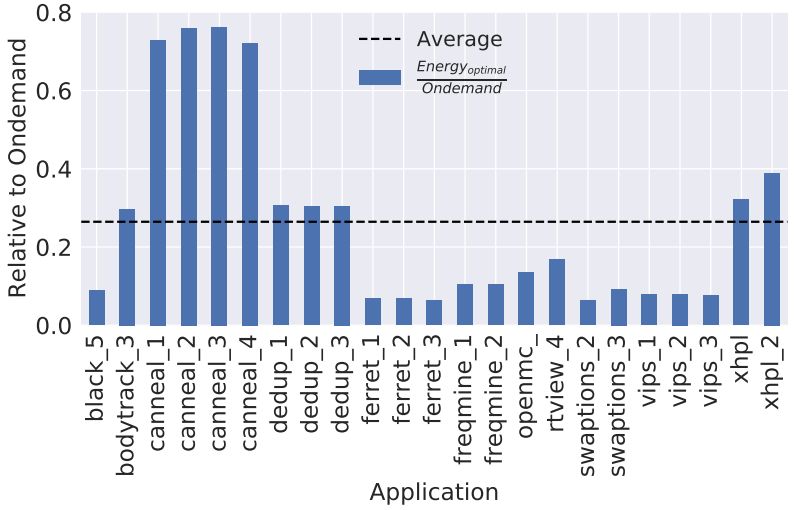
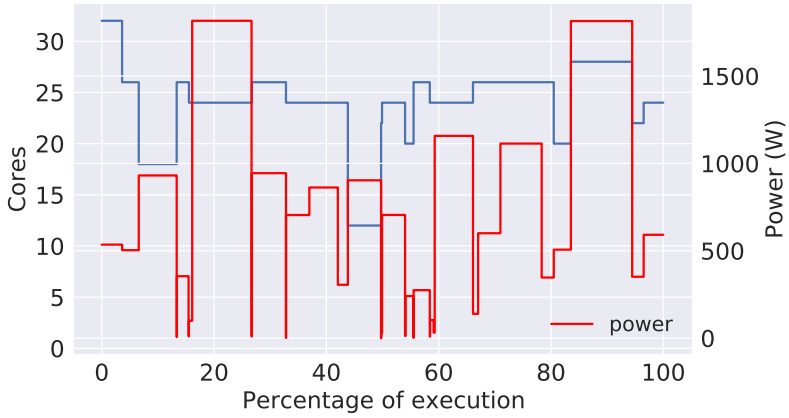


Figure 5.9: Optimal phase splitting energy vs. on-demand governor on Linux: relative energy comparison for applications with different input sizes for the best choice case. Lower is better.

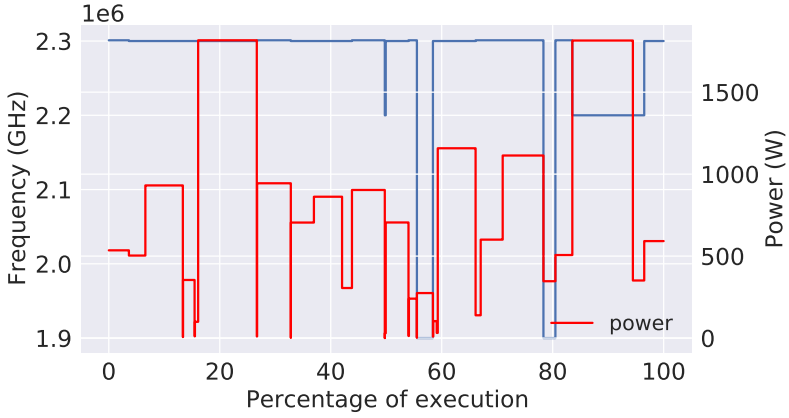
be transposed to the target hardware for power consumption estimation. In addition, it is important to compute these metrics using a combination of hardware performance counters that allow for optimal real-time phase splitting. This metric can improve current DVFS algorithms by providing them with a workload metric to improve energy savings.

Looking at programs' characterization, relevant metrics can be utilized to construct a profile or an application fingerprint, illustrating its behavior across various parameters other than frequency and core count. As such, the fingerprint can be used as an alternative method to estimate the phase division leading to minimal energy consumption. This is a constructive method that can follow, with a growing granularity, variations in the fingerprint. Indeed, there is a relationship between the phase distribution and the fingerprint. For instance, The Section 5.9 displays the optimal phase distribution obtained by the proposed method superposed to the power profiles collected during the execution for a specific application. We deliberately chose the number of phases to highlight this correlation. Observe the changes in power profiles with variations in frequency and number of cores.

Some profiles, within specific parameter limits, exhibit consistent patterns, facilitating the identification of phases. Additionally, we observed that pertinent metrics can be amalgamated through a set of equations, yielding a distinctive representation. These equations may assume diverse forms, contingent on the application's characteristics and the chosen counters. For instance, a straightforward equation could entail the summation of all counters, whereas a more intricate one might involve a weighted sum or a combination with distinct scaling factors. Furthermore, profiles can be leveraged in different ways, for example, by comparing them to a set of known phase divisions or utilizing machine learning techniques to detect such patterns. By identifying these patterns, we can determine the phase divisions to be employed by the optimization method. Finally, by discerning the relevant patterns and trends, we can approximate the application's behavior for other configuration variants.



(a) Black-Scholes application using 35 phases (Power profile config using 25 cores and 2.3GHz frequency).



(b) Bodytrack application using 35 phases (Power profile config using 25 cores and 2.3GHz frequency).

Figure 5.10: Power profiles and the optimal phase distribution for a given configuration.

5.10. Application Characterization, Modeling, and Behavioral Clustering Based on Fingerprint

Based on the data we can collect, we can correctly model the behavior of applications and provide performance figures in time or execution percentage representations, referred to as fingerprints. These fingerprint representations can be used to model, characterize, compare, and cluster applications according to performance and behavioral parameters.

5.10.1 Defining a Fingerprint Metric

To cluster applications, we first need a way to compare two programs. We define a new variable to compute a fingerprint for the program, which should appear similar when executing the same program under different conditions and inputs. Considering the simplest model of a program as a Turing machine, where everything can be done with a tape of memory and a set of rules, we empirically define the variable input size as described in Eq. (5.1). On real computers, this is not too far from reality, as most computations and input/output operations pass through memory. Analyzing the relationship between the total number of instructions and memory instructions provides a good fingerprint.

$$I_{sz} = \frac{I}{I_m} \tag{5.1}$$

where I_{sz} is the input size, I is the number of instructions executed, and I_m is the number of memory instructions executed.

We observe that this variable demonstrates the properties we are looking for, producing similar results for the same program with different input sizes and environments. This can be used to identify programs and to observe similarities between different applications.

To compute the distance between two programs, we use the Canberra metric Jurman (2009), described in Eq. (5.2).

$$d(p,q) = \sum_{i=1}^{n} \frac{|p_i - q_i|}{|p_i| + |q_i|} \tag{5.2}$$

where p and q are n-dimensional vectors.

5.10.2 Clustering Applications Based on Fingerprint Metric

We computed the input size for all 30 applications in the Polybench benchmark suite Gonzalez (2021) with 3 different input sizes. Each program was run 15 times, with a sampling rate of 0.01 seconds, collecting the total number of instructions, number of load and store instructions, and number of floating-point operations, along with some software counters.

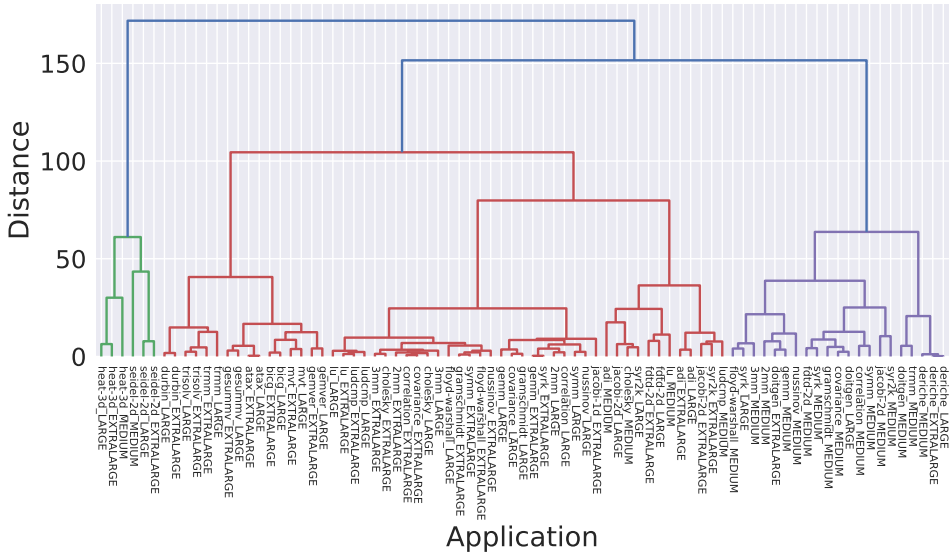


Figure 5.11: Dendrogram of fingerprint according to Eq. (5.1) for 30 applications in the Polybench benchmark suite.

After post-processing the collected data, we compute the input size and perform hierarchical clustering using Ward's linkage method Murtagh (2011), which minimizes the total within-cluster variance. The results of the clustering are shown in Fig. 5.12 and the dendrogram in Fig. 5.11.

From the dendrogram, we can infer the similarity between applications. We chose the number of clusters that maximize the occurrence of the same program with different input sizes in the same cluster; in this case,

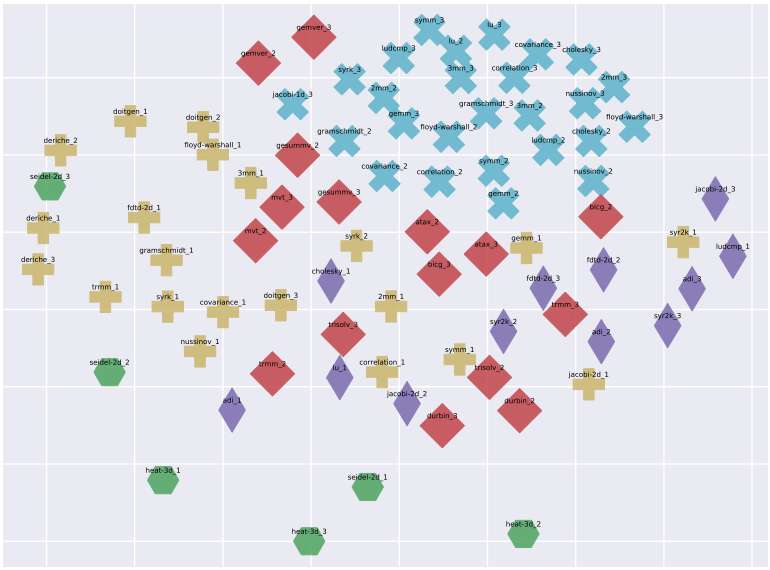


Figure 5.12: Spring force graph of Canberra distance according to Eq. (5.1) for 30 applications in the Polybench benchmark suite.

five clusters were identified.

We observed that as the input size grows, program behavior tends toward a specific characteristic fingerprint, but for small input sizes, some variations exist, leading to the same program being classified in more than one cluster. This can also happen if specific parts of the program are triggered with specific inputs, resulting in it belonging to multiple clusters.

To obtain an overall classification of programs, we consider the frequency of each program's appearance in the clusters and classify it in the cluster where it appears most often. The clusters are as follows:

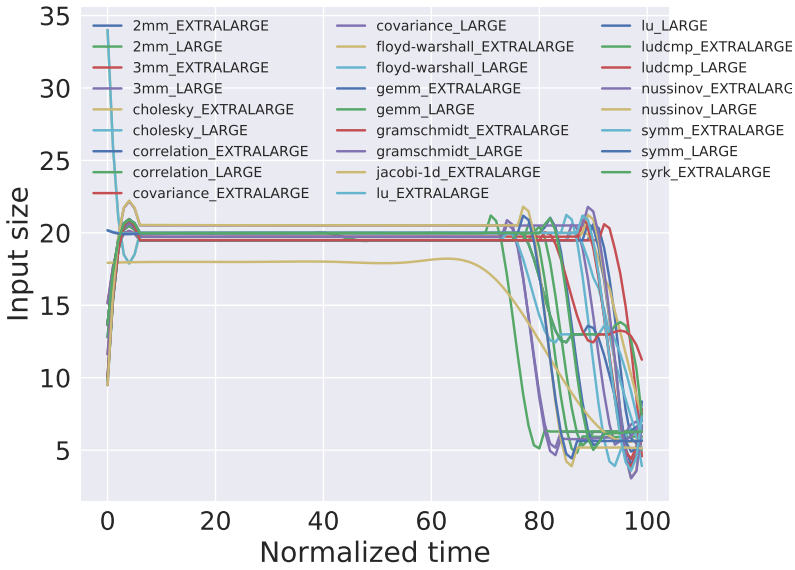
- Cluster 1: 2mm, 3mm, cholesky, correlation, covariance, floyd-warshall, gemm, gramschmidt, lu, ludcmp, nussinov, symm
- Cluster 2: deriche, doitgen, syrk
- Cluster 3: adi, fdtd-2d, jacobi-2d, syr2k
- Cluster 4: atax, bicg, durbin, gemver, gesummv, mvt, trisolv, trmm

- Cluster 5: heat-3d, seidel-2d

In Figure 5.12, we display the clusters using the Spring Force algorithm, where each program with a specific input is a node, and the edge weight is the Canberra distance. For better visualization, the input names were replaced by numbers: EXTRALARGE is 3, LARGE is 2, and MEDIUM is 1. From this graph, we see that clustering is well partitioned, and each cluster is distinct. It is also interesting to note that applications in the cluster represented by the circle symbol are more separated, indicating they were classified this way because they did not fit well into any other cluster.

To understand the behavior of the variable input size for each cluster, we plotted it for the applications in clusters 1 and 2 in Figs. 5.13 and 5.14.

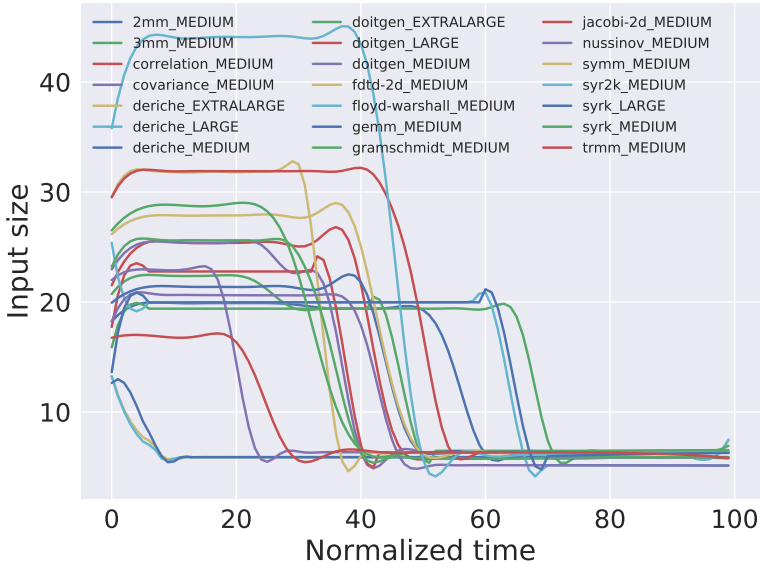
Figure 5.13: Input size - Cluster 1



From these fingerprints, we can infer why certain behaviors were classified in the same cluster.

In cluster 1, Figure 5.13, all applications show similar behavior with approximately the same amplitude.

Figure 5.14: Input size - Cluster 2



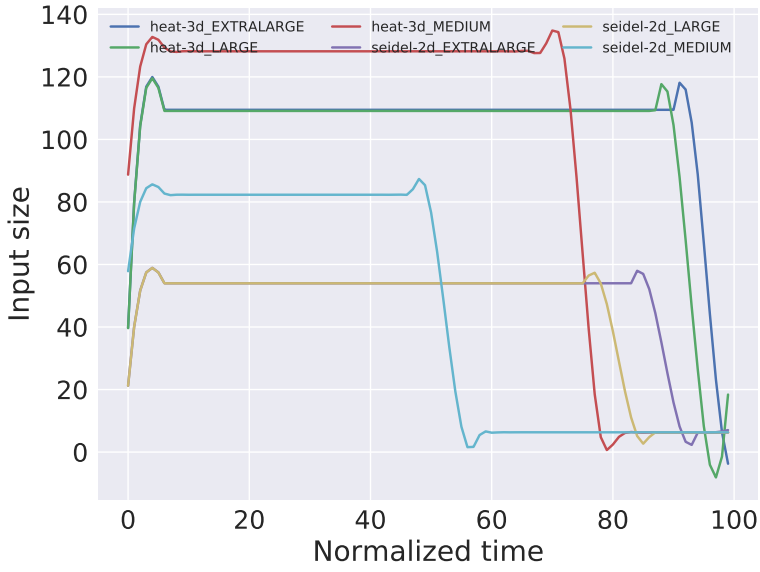


Figure 5.15: Input size - Cluster 3

In Figure 5.14, we observe that curves with similar shapes, regardless of the scale on the vertical and horizontal axes, were also classified in the same cluster. This is the desired behavior, as it demonstrates that our tool can compare and classify programs according to their fingerprint behavioral representation.

This demonstrates that fingerprinting is an excellent tool for better modeling and characterization of programs, regardless of their size and complexity. Additionally, this technique can be extended to various other performance parameters related to behavior depending on the processor's characteristics, as detailed in the following sections.

To conclude this study, we also clustered applications based on their floating-point behavior. For this, we only used the counter corresponding to floating-point operations. Figure 5.16 shows the dendrogram, and Figure 5.17 displays the spring force graph plot.

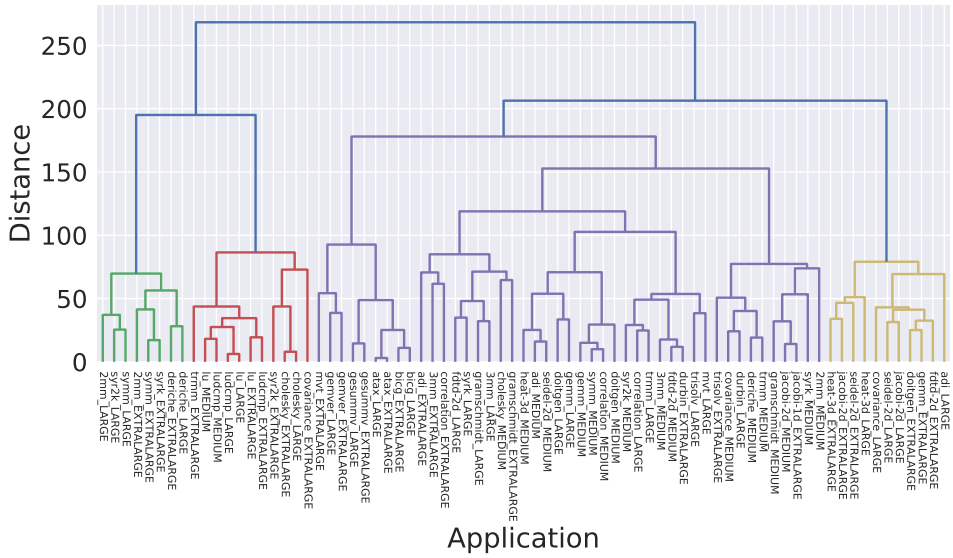


Figure 5.16: Dendrogram of fingerprint according to number of floating-point operations per second for 30 applications in the Polybench benchmark suite.

Figures 5.18 to 5.20 show the behavior within the same cluster for floating-point operations. For this classification, the number of clusters

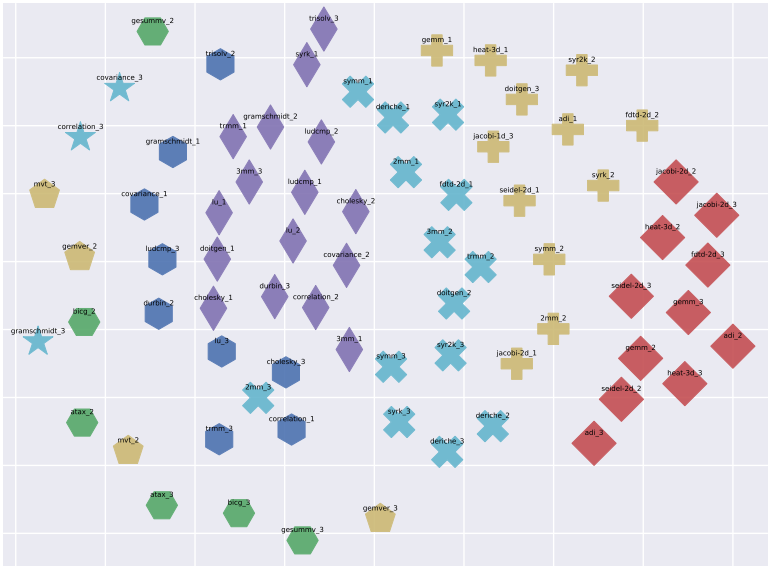


Figure 5.17: Spring force graph of Canberra distance for floating-point operations per second for 30 applications in the Polybench benchmark suite.

was 8.

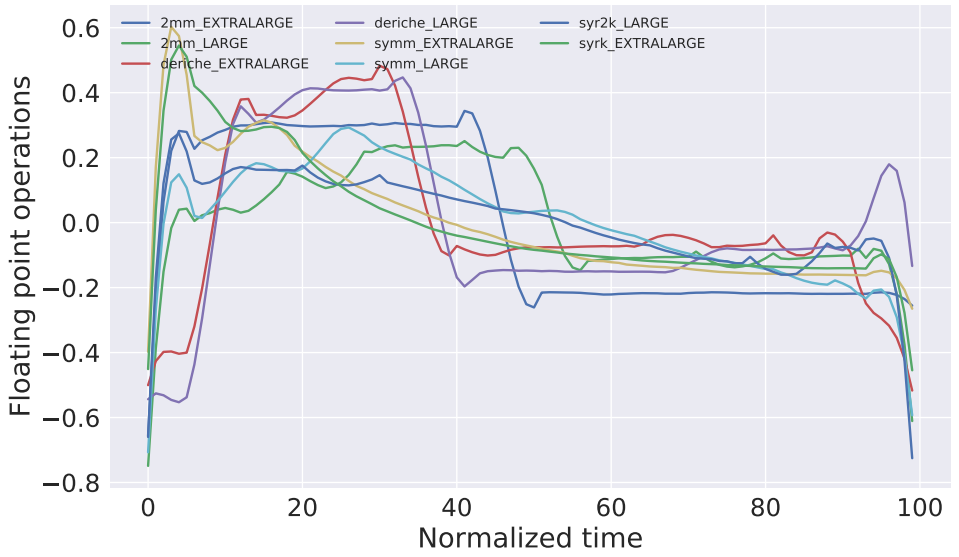


Figure 5.18: Floating point - Cluster 1

Here, we observe again that applications with similar shapes were classified into the same clusters.

With this methodology, we can develop generic energy models for each application cluster. By leveraging the fingerprint representations and clustering techniques, we can capture the unique characteristics and behaviors of different applications. Additionally, we can utilize various input metrics to model energy consumption, ensuring that there is at least one metric that works exceptionally well for accurately predicting energy usage. This approach provides a robust framework for modeling and optimizing energy efficiency across a wide range of applications, ultimately contributing to more efficient and sustainable high-performance computing environments.

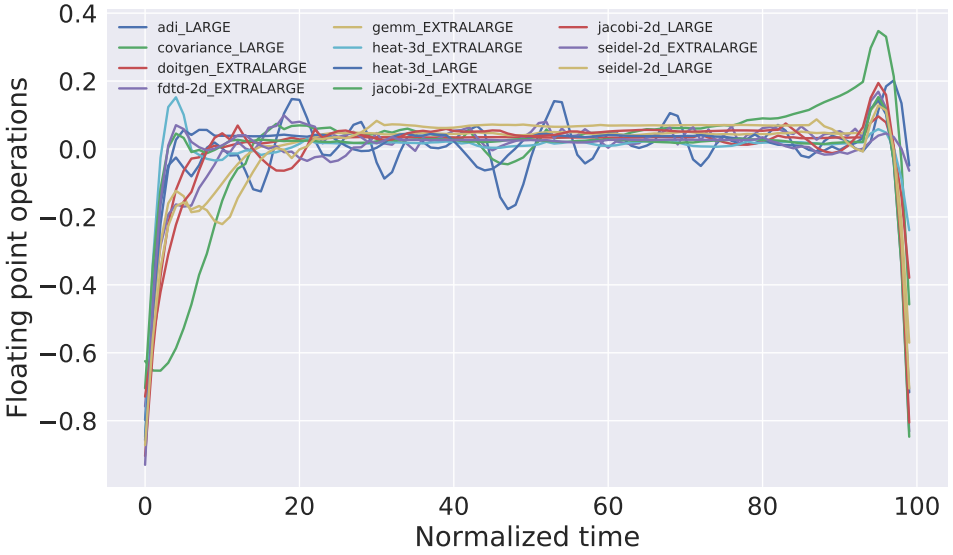


Figure 5.19: Floating point - Cluster 2

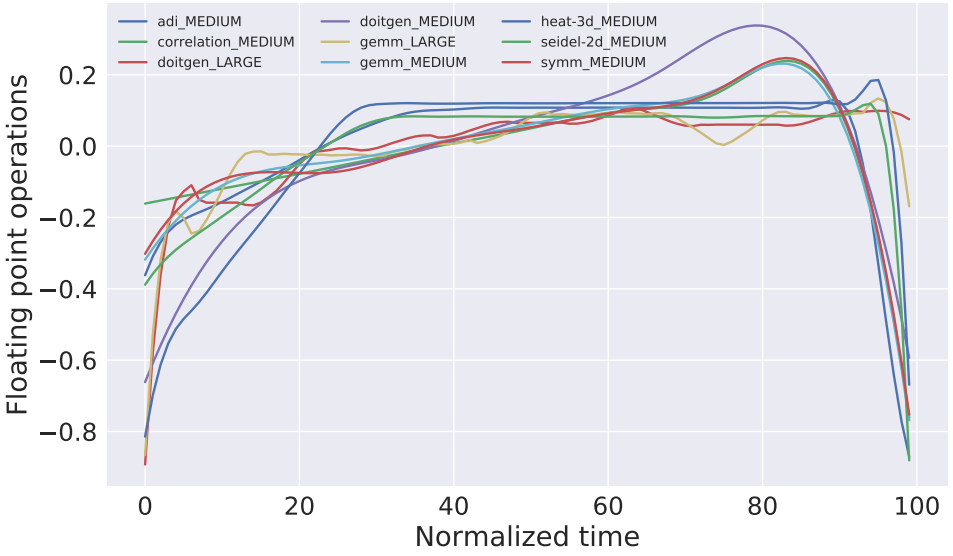


Figure 5.20: Floating point - Cluster 3

CHAPTER 6

Conclusions and future work

In this chapter, we deal with the conclusions about the effectiveness of the frameworks and the advantages and disadvantages of using models and algorithms with additional information on applications and architectures for energy optimization. We also discussed what could be improved in each approach and framework.

6.1. Introduction

This thesis presents a comprehensive approach to modeling HPC applications, featuring a low-overhead framework for data collection, an analytical model for energy consumption, and a heuristic algorithm for multi-phase applications. Each of the methods presented has potential improvements that will be discussed below.

6.2. Pascal Suite Framework

The Pascal Suite framework introduces practical and user-friendly tools for measuring and analyzing the efficiency of parallel applications. The tool focuses on observing energy consumption and scalability, implementing features that enable analysis at hierarchical levels of the program's inner parts. It also simplifies the comparison of application runs under different configurations, aiding developers in targeting software optimization efforts.

The tool adds minimal intrusion to the program's performance measurement under analysis, which is crucial for understanding the program's behavior.

The results indicate that the fingerprint module API has an overhead similar to or lower than other low-level APIs, with the added advantage of high abstraction and simplified configuration. With just a few lines of code, it is possible to configure and gather performance counter data.

The developed fingerprint module also provided a way to compute similarities between different programs or the same program with different inputs. This can be useful for reducing application spaces for benchmarks, as demonstrated in Polybench clustering, and for analyzing parameter behavior, providing insights to identify potential bottlenecks.

Future work will focus on evolving this tool to include the ability to predict speedup and efficiency from a few samples using state-of-the-art prediction models from the literature Alex (2020); Silva (2020). The goal is to present the general behavior of the program and reduce the execution time necessary for comprehensive analysis. Additionally, we plan to include features that allow the observation of parallel applications in distributed environments using the Message Passing Interface (MPI) standard.

6.3. Application-Energy Model

The proposed energy model based on frequency and the number of cores for a full shared memory system can serve as a basis for DVFS and DPM optimization problems that include both frequency and active cores. It can also be used to analyze the contribution of each parameter (e.g., parallelism level) to energy consumption.

Results from three HPC benchmarks running on a cluster demonstrate the potential of the proposed model. While consuming less energy than traditional machine learning approaches, it can serve as a basis for DVFS and DPM algorithms, as shown in Section 4.12, achieving average energy savings of about 12% up to 69%. Prior knowledge of the application's performance can reveal significant insights, such as parallel speedups, which are difficult to estimate with run-time techniques based on DVFS.

A limitation of the proposed model is the need for information about the application's input size, which can be complex to derive. A potential solution is to precisely define what constitutes input size, using a common variable for all applications, such as throughput. Future work will explore the various analyses possible with the equation and develop more advanced DVFS models. For instance, identifying different phases of the target program could enable more fine-grained adjustments to frequency and the number of active cores, further improving the results presented here.

Another important aspect often overlooked is the number of processing cores to be used by a parallel program. This choice is typically left to the user, which, as shown in this thesis, is not a trivial decision.

6.4. Phase Division Approach

The main conclusion of this work is that in the HPC environment, not many phases are needed to achieve optimal energy consumption, even though finding the optimal phase divisions has clear advantages. An average maximum of 35 phases was sufficient for the three benchmarks, covering a wide range of HPC applications.

It was also observed that there is still significant potential for optimizing DVFS algorithms, as we achieved an average of 38% energy savings

compared to the Linux OnDemand governor. Additionally, a relationship between application behavior and phase locations was noted, independent of their number.

In this work, we propose a phase division algorithm that can improve other power management techniques in several ways. By providing a detailed analysis of the energy impact of an application under different execution conditions, the phase division algorithm can help identify energy-intensive phases. This information can be used to characterize and model applications according to the target CPU, optimizing the application's execution by focusing on the most energy-intensive phases.

The algorithm can also identify idle phases, which can then be managed by power-saving techniques such as DPM or slack recovery. Particularly energy-intensive phases, such as data loading or computation, can be identified, characterized, and targeted for optimization through task allocation or Crown scheduling.

By determining the phases of the application, the proposed algorithm provides a granular view of power consumption, allowing finer control of power management techniques. This can lead to more effective energy savings and improved energy efficiency.

Furthermore, the algorithm can provide information on the optimal number of phases and intervals, which can be used to optimize scheduling algorithms, thereby reducing energy consumption without affecting system performance.

6.5. General

In summary, this thesis provides a holistic approach to energy optimization in HPC applications. The integration of practical tools, robust models, and innovative algorithms offers a pathway to significant energy savings and enhanced efficiency. By addressing both theoretical and practical aspects of energy consumption, this work contributes to the broader goal of sustainable and efficient high-performance computing.

6.6. Extensibility for Future Research

While this thesis presents significant advancements in optimizing energy efficiency for cluster-based HPC systems, it is primarily focused on shared memory architectures. This focus, while allowing for deep exploration and refinement of energy optimization techniques within this specific context, also represents a limitation when considering the broader landscape of HPC. For instance, the methods and models developed in this work have not been directly applied to distributed platforms such as those utilizing OpenMPI, which manage communication and computation across multiple nodes in a network. Extending this research to include distributed HPC platforms involves adapting the proposed algorithms to handle the complexities of inter-node communication and synchronization, which are critical in such environments.

Additionally, the thesis does not address GPU-based platforms, which have become increasingly important in high-performance computing due to the increasing integration of artificial intelligence models like High Performance Computing (HPC), particularly for workloads that benefit from massive parallelism. Integrating GPU-based optimization into the existing framework would require accounting for the unique power and performance characteristics of GPUs, as well as their interaction with CPUs in heterogeneous computing environments.

Finally, a significant portion of large data centers operate using containerized applications, which often lack direct control over the CPU. This limitation could hinder the optimization techniques developed in this thesis. However, these challenges can be addressed through enterprise-level solutions. By implementing CPU management at the infrastructure level, enterprises can apply energy-saving techniques transparently, allowing containerized applications to benefit from these optimizations without compromising their inherent advantages.

The extensibility of this work is evident in its potential applications beyond traditional HPC environments. Future research can build upon the foundation laid in this thesis to explore new directions:

- Application to Distributed HPC Platforms: Adapting the algorithms and models for use in distributed HPC platforms, such as those using OpenMPI, presents a significant opportunity for extending

this work. This would require addressing the challenges of inter-node communication, synchronization, and the varying energy profiles of nodes in a distributed system.

- GPU-Based Optimization: With the growing importance of GPUs in HPC, particularly for AI and machine learning workloads, future research could focus on integrating GPU-based optimization into the existing framework. This would involve developing energy models specific to GPU workloads and understanding the interaction between CPU and GPU energy consumption in heterogeneous systems.

- Exploration of Hybrid Architectures: As HPC systems continue to evolve, incorporating hybrid architectures that combine CPUs, GPUs, and possibly other accelerators, there is an opportunity to extend the framework to manage energy efficiency across these diverse architectures.

Bibliography

Ishfag, A.; Sanjay, R. *Handbook of Energy-Aware and Green Computing*; Chapman & Hall/CRC: London, England, 2012; Volume 1, pp. 702–713.

Dayarathna, M.; Wen, Y.; Fan, R. Data Center Energy Consumption Modeling: A Survey. *IEEE Commun. Surv. Tutor.* **2016**, *18*, 732–794. [CrossRef]

Corcoran, P.; Andrae, A. *Emerging Trends in Electricity Consumption for Consumer ICT*; National University of Ireland: Galway, Ireland, 2013; pp. 1–56.

Mathew, V.; Sitaraman, R. K.; Shenoy, P. Energy-aware load balancing in content delivery networks. In Proceedings of the 2012 Proceedings IEEE INFOCOM, Orlando, FL, USA, 25–30 March 2012; pp. 954–962.

Rivoire, S.; Shah, M.A.; Ranganathan, P.; Kozyrakis, C.; Meza, J. Models and Metrics to Enable Energy-Efficiency Optimizations. *Computer* **2007**, *40*, 39–48. [CrossRef]

Buyya, R.; Vecchiola, C.; Selvi, S.T. *Mastering Cloud Computing*; Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2013; pp. 3–27

Poess, M.; Nambiar, R.O. Energy cost, the key challenge of today's data centers. *Proc. VLDB Endow.* **2008**, *1*, 1229–1240. [CrossRef]

Gao, Y.; Guan, H.; Qi, Z.; Wang, B.; Liu, L. Quality of service aware

power management for virtualized data centers. *J. Syst. Archit.* **2013**, *59*, 245–259. [CrossRef]

Fan, X.; Weber, W.D.; Barroso, L.A. Power provisioning for a warehouse-sized computer. *ACM SIGARCH Comput. Archit. News* **2007**, *35*, 13–23. [CrossRef]

Barroso, L.A.; Hölzle, U. The Case for Energy-Proportional Computing. *Computer* **2007**, *40*, 33–37. [CrossRef]

Malladi, K.T.; Nothaft, F.A.; Periyathambi, K.; Lee, B.C.; Kozyrakis, C.; Horowitz, M. Towards energy-proportional datacenter memory with mobile DRAM. In Proceedings of the 2012 39th Annual International Symposium on Computer Architecture (ISCA), Portland, OR, USA, 9–13 June 2012; pp. 37–48.

Rotem, E.; Naveh, A.; Ananthakrishnan, A.; Weissmann, E.; Rajwan, D. Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge. *IEEE Micro* **2012**, *32* 20–27. [CrossRef]

Brown, L.; Moore, R.; Li, D.S.; Yu, L.; Keshavamurthy, A.; Pallipadi, V. ACPI in Linux. *Symposium* **2005**, *51*, 1–51.

Hackenberg, D.; Schone, R.; Ilsche, T.; Molka, D.; Schuchart, J.; Geyer, R. An Energy Efficiency Feature Survey of the Intel Haswell Processor. In Proceedings of the 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, Hyderabad, India, 25–29 May 2015; pp. 896–904.

Intel. *12th Generation Intel ® Core™ Processors*; Intel: Santa Clara, CA, USA, 2020; pp. 420–430

Shuja, J.; Madani, S.A.; Bilal, K.; Hayat, K.; Khan, S.U.; Sarwar, S. Energy-efficient data centers. *Computing* **2012**, *94*, 973–994. [CrossRef]

Benini, L.; Bogliolo, A.; De Micheli, G. A survey of design techniques for system-level dynamic power management. *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.* **2000**, *8*, 299–316. [CrossRef]

Merkel, A.; Bellosa, F. Balancing power consumption in multiprocessor

systems. *ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst.* **2006**, *40*, 403–4014.

Roy, S.; Rudra, A.; Verma, A. An energy complexity model for algorithms. In Proceedings of the 4th conference on Innovations in Theoretical Computer Science, New York, NY, USA, 9–12 January 2013.

Weaver, V.M.; McKee, S.A. Can hardware performance counters be trusted? In Proceedings of the 2008 IEEE International Symposium on Workload Characterization, Seattle, WA, USA, 14–16 September 2008; pp. 141–150.

Weaver, V.M.; Terpstra, D.; Moore, S. Non-determinism and overcount on modern hardware performance counter implementations. In Proceedings of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Austin, TX, USA, 21–23 April 2013; pp. 215–224.

Das, S.; Werner, J.; Antonakakis, M.; Polychronakis, M.; Monrose, F. SoK: The Challenges, Pitfalls, and Perils of Using Hardware Performance Counters for Security. In Proceedings of the 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 19–23 May 2019; pp. 20–38.

Mc Guire, N.; Okech, P.; Schiesser, G. Analysis of Inherent Randomness of the Linux Kernel. In Proceedings of the Eleventh RealTime Linux Workshop, Dresden, Germany, 28–30 September 2009.

Ramos, V.; Valderrama, C.; Xavier de Souza, S.; Manneback, P. An Accurate Tool for Modeling, Fingerprinting, Comparison, and Clustering of Parallel Applications Based on Performance Counters. In Proceedings of the IEEE International Parallel and Distributed Processing, Rio de Janeiro, Brazil, 20–24 May 2019; pp. 797–804.

Silva-de Souza, W.; Iranfar, A.; Bráulio, A.; Zapater, M.; Xavier-de Souza, S.; Olcoz, K.; Atienza, D. Containergy—A Container-Based Energy and Performance Profiling Tool for Next Generation Workloads. *Energies* **2020**, *13*, 2162. [CrossRef]

Shao, Y.S.; Brooks, D. Energy characterization and instruction-level energy model of Intel's Xeon Phi processor. In Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED), Beijing, China, 4–6 September 2013; pp. 389–394.

Lewis, A.; Ghosh, S.; Tzeng, N.F. Run-time energy consumption estimation based on workload in server systems. In Proceedings of the 2008 Conference on Power Aware Computing and Systems, San Diego, CA, USA, 8–10 December 2008; pp. 3–4.

Mills, B.; Znati, T.; Melhem, R.; Ferreira, K.B.; Grant, R.E. Energy Consumption of Resilience Mechanisms in Large Scale Systems. In Proceedings of the 2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Turin, Italy, 12–14 February 2014; pp. 528–535.

Feng, W.c. Making a Case for Efficient Supercomputing. *Queue* **2003**, *1*, 54–64. [CrossRef]

Sarwar, A. Cmos power consumption and cpd calculation. In *Proceeding: Design Considerations for Logic Products*; Texas Instruments: Dallas, TX, USA, 1997.

Butzen, P.; Ribas, R. *Leakage Current in Sub-Micrometer CMOS Gates*; Universidade Federal do Rio Grande do Sul: Porto Alegre, Brazil, 2007; pp. 1–30.

Amdahl, G.M. Validity of the single processor approach to achieving large scale computing capabilities. In Proceedings of the Spring Joint Computer Conference on—AFIPS '67 (Spring), New York, NY, USA, 18–20 April 1967.

Eyerman, S.; Eeckhout, L. Modeling critical sections in Amdahl's law and its implications for multicore design. In Proceedings of the 37th Annual International Symposium on Computer Architecture—ISCA '10, New York, NY, USA, 19–23 June 2010.

Gustafson, J.L. Reevaluating Amdahl's law. *Commun. ACM* **1988**, *31*, 532–533. [CrossRef]

Seel, N.M. *Encyclopedia of the Sciences of Learning*; Springer: Berlin/Heidelberg, Germany, 1988; pp. 223–242.

Roy, P.; Mahapatra, G.S.; Dey, K.N. Forecasting of software reliability using neighborhood fuzzy particle swarm optimization based novel neural network. *IEEE/CAA J. Autom. Sin.* **2019**, *6*, 1365–1383. [CrossRef]

Zhu, W.; Liu, X.; Xu, M.; Wu, H. Predicting the results of RNA molecular specific hybridization using machine learning. *IEEE/CAA J. Autom. Sin.* **2019**, *6*, 1384–1396. [CrossRef]

Rivoire, S.; Ranganathan, P.; Kozyrakis, C. A comparison of high-level full-system power models. In Proceedings of the 2008 Conference on Power Aware Computing and Systems, San Diego, CA, USA, 8–10 December 2008; pp. 1–5.

Usman, S.; Khan, S.U.; Khan, S. A comparative study of voltage/frequency scaling in NoC. In Proceedings of the IEEE International Conference on Electro-Information Technology, Rapid City, SD, USA, 9–11 May 2013; pp. 1–5.

Paolillo, A. Optimisation of Performance Metrics of Embedded Hard Real-Time Systems using Software/Hardware Parallelism, Ph.D. Thesis, Université libre de Bruxelles, Brussels, Belgium, 2018.

Kim, D.H.; Imes, C.; Hoffmann, H. Racing and Pacing to Idle: Theoretical and Empirical Analysis of Energy Optimization Heuristics. In Proceedings of the 2015 IEEE 3rd International Conference on Cyber-Physical Systems, Networks, and Applications, Hong Kong, China, 19–21 August 2015; pp. 78–85.

Fu, C.; Chau, V.; Li, M.; Xue, C.J. Race to idle or not: Balancing the memory sleep time with DVS for energy minimization. *J. Comb. Optim.* **2018**, *35*, 860–894. [CrossRef]

Rauber, T.; Rünger, G.; Schwind, M.; Xu, H.; Melzner, S. Energy measurement, modeling, and prediction for processors with frequency scaling. *J. Supercomput.* **2014**, *70*, 1451–1476. [CrossRef]

Goel, B.; McKee, S.A. A Methodology for Modeling Dynamic and Static Power Consumption for Multicore Processors. In Proceedings of the IEEE International Parallel and Distributed Processing Symposium, Chicago, IL, USA, 23–27 May 2016; pp. 273–282.

Du, Z.; Ge, R.; Lee, V.W.; Vuduc, R.; Bader, D.A.; He, L. Modeling the Power Variability of Core Speed Scaling on Homogeneous Multicore Systems. *Sci. Program.* **2017**, *2017*, 1–13. [CrossRef]

Gonzalez, R.; Gordon, B.; Horowitz, M. Supply and threshold voltage scaling for low power CMOS. *IEEE J. Solid-State Circuits* **1997**, *32*, 1210–1216. [CrossRef]

Silva, V.R.G.; Furtunato, A.F.A.; Georgiou, K.; Sakuyama, C.A.V.; Eder, K.; Xavier-de Souza, S. Energy-Optimal Configurations for Single-Node HPC Applications. In Proceedings of the 2019 International Conference on High Performance Computing & Simulation (HPCS), Dublin, Ireland, 15–19 July 2019; pp. 448–454.

Kumar, V.; Gupta, A. Analyzing Scalability of Parallel Algorithms and Architectures. *J. Parallel Distrib. Comput.* **1994**, *22*, 379–391. [CrossRef]

Oliveira, V.H.F.; Furtunato, A.F.A.; Silveira, L.F.; Georgiou, K.; Eder, K.; Xavier-de Souza, S. Application Speedup Characterization. In Proceedings of the ACM/SPEC International Conference on Performance Engineering, Berlin, Germany, 9–13 April 2018; pp. 43–44.

Smola, A.J.; Schölkopf, B. A tutorial on support vector regression. *Stat. Comput.* **2004**, *14*, 199–222. [CrossRef]

Kitts, B. Regression Trees Lecture. *Data Min.* **2006**, 6–7.

Altman, N.S. An Introduction to Kernel and Nearest-Neighbor Nonparametric Regression. *Am. Stat.* **1992**, *46*, 175–185.

Murtagh, F. Multilayer perceptrons for classification and regression. *Neurocomputing* **1991**, *2*, 183–197. [CrossRef]

Gao, S.; Zhou, M.; Wang, Y.; Cheng, J.; Yachi, H.; Wang, J. Dendritic

Neuron Model With Effective Learning Algorithms for Classification, Approximation, and Prediction. *IEEE Trans. Neural Netw. Learn. Syst.* **2019**, *30*, 601–614. [CrossRef]

Schwenkler, T.; Deutschland, S. Intelligent Platform Management Interface. In *Sicheres Netzwerkmanagement*; Springer: Berlin/Heidelberg, Germany, 2006; pp. 169–207.

Bienia, C.; Kumar, S.; Singh, J.P.; Li, K. The PARSEC benchmark suite. In Proceedings of the 17th international conference on Parallel architectures and compilation techniques—PACT '08, New York, NY, USA, 25–29 October 2008.

Romano, P.K.; Horelik, N.E.; Herman, B.R.; Nelson, A.G.; Forget, B.; Smith, K. OpenMC: A state-of-the-art Monte Carlo code for research and development. *Ann. Nucl. Energy* **2015**, *82*, 90–97. [CrossRef]

Dongarra, J.J. The LINPACK Benchmark: An explanation. In Proceedings of the 1st International Conference on Supercomputing, Athens, Greece, 8–12 June 1987.

Pedregosa, F; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; et al. Scikit-learn: Machine Learning in {P}ython. *J. Mach. Learn. Res.* **2011**, *12*, 2825–2830.

Royer, C.W.; O'Neill, M.; Wright, S.J. A Newton-CG algorithm with complexity guarantees for smooth unconstrained optimization. *Math. Programm.* **2020**, *180*, 451–488.

Tibor Horyath and Kevin Skadron. Multi-mode energy management for multi-tier server clusters. *Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT*, 270–279, 2008.

John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. *ACM SIGARCH Computer Architecture News*, 41(3):559, 2013.

Stephane Eranian. Perfmon2: a standard performance monitoring interface for Linux. *Slides, perfmon2 overview*, 2008.

Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring energy consumption for short code paths using RAPL. *ACM SIGMETRICS Performance Evaluation Review*, 40(3):13, 2012.

Reza Zamani and Ahmad Afsahi. A study of hardware performance monitoring counter selection in power modeling of computing systems. *2012 International Green Computing Conference, IGCC 2012*, 2012.

IPMI Configuration User Guide. 2017(November), 2013.

Fionn Murtagh and Pierre Legendre. Ward's Hierarchical Clustering Method: Clustering Criterion and Agglomerative Algorithm. (June):1–20, 2011.

PJ Mucci, Shirley Browne, Christine Deane, and George Ho. PAPI: A portable interface to hardware performance counters. *Proceedings of the department of defense HPCMP users group conference*, 32:7–10, 1999.

Jianwen Luo, Kui Ying, Ping He, and Jing Bai. Properties of Savitzky-Golay digital differentiators. *Digital Signal Processing: A Review Journal*, 15(2):122–136, 2005.

Rick Kufrin. Perfsuite: An accessible, open source performance analysis environment for linux. *Dans Presented at The 6th International Conference on Linux Clusters: The HPC Revolution*, 151(April):5, 2005.

Andreas Knüpfer and Christian Rössel. Score-P – A Joint Performance Measurement Run-Time Infrastructure for. 1–12, 2011.

Giuseppe Jurman, Samantha Riccadonna, Roberto Visintainer, and Cesare Furlanello. Canberra distance on ranked lists. *Proceedings, Advances in Ranking-NIPS 09 Workshop*, pages 22–27, 2009.

Houjun Hang, Xing Yao, Qingqing Li, and Michel Artiles. Cubic B-Spline Curves with Shape Parameter and Their Applications. *Mathematical Problems in Engineering*, 2017:1–8, 2017.

Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide. 3(253665):1–1386, 2013.

Huck, K.; Malony, A.; Shende, S.; Morris, A. Scalable, Automated Performance Analysis with TAU and PerfExplorer. In Proceedings of the PARCO, Aachen, Germany, 4–7 September 2007; Volume 15, pp. 629–636.

Islam, T.; Ayala, A.; Jensen, Q.; Ibrahim, K. Toward a Programmable Analysis and Visualization Framework for Interactive Performance Analytics. In Proceedings of the IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools), Denver, CO, USA, 17 November 2019; pp. 70–77. [CrossRef]

Weber, M.; Ziegenbalg, J.; Wesarg, B. Online Performance Analysis with the Vampir Tool Set. In *Tools for High Performance Computing 2017, Proceedings of the 11th International Workshop on Parallel Tools for High Performance Computing, Dresden, Germany, 11–12 September, 2017* ; Springer International Publishing: Cham, Switzerland, 2019; pp. 129–143. [CrossRef]

Bergel, A.; Bhatele, A.; Boehme, D.; Gralka, P.; Griffin, K.; Hermanns, M.A.; Okanović, D.; Pearce, O.; Vierjahn, T. *Visual Analytics Challenges in Analyzing Calling Context Trees*; Springer: Cham, Switzerland, 2019; pp. 233–249. [CrossRef]

Huck, K.; Malony, A. PerfExplorer: A Performance Data Mining Framework For Large-Scale Parallel Computing. In Proceedings of the SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, Seattle, WA, USA, 12–18 November 2005; p. 41. [CrossRef]

Geimer, M.; Wolf, F.; Wylie, B.; Ábrahám, E.; Becker, D.; Mohr, B. The Scalasca performance toolset architecture. *Concurr. Comput. Pract. Exp.* **2010**, *22*, 702–719. [CrossRef]

Shende, S.S.; Malony, A.D. The Tau Parallel Performance System. *Int. J. High Perform. Comput. Appl.* **2006**, *20*, 287–311. [CrossRef]

Adhianto, L.; Banerjee, S.; Fagan, M.; Krentel, M.; Marin, G.; Mellor-Crummey, J.; Tallent, N.R. HPCTOOLKIT: Tools for performance analysis of optimized parallel programs. *Concurr. Comput. Pract. Exp.* **2010**, *22*, 685–701. [CrossRef]

Miller, B.; Callaghan, M.; Cargille, J.; Hollingsworth, J.; Irvin, R.; Karavanic, K.; Kunchithapadam, K.; Newhall, T. The Paradyn parallel performance measurement tool. *Computer* **1995**, *28*, 37–46. [CrossRef]

Galobardes, E.C. *Automatic Tuning of HPC Applications. The Periscope Tuning Framework*; Shaker: Herzogenrath, Germany, 2015.

Pillet, V.; Labarta, J.; Cortes, T.; Girona, S. PARAVER: A Tool to Visualize and Analyze Parallel Code. In Proceedings of the WoTUG-18: Transputer and Occam Developments, Manchester, UK, 9–13 April 2007.

Brink, S.; Lumsden, I.; Scully-Allison, C.; Williams, K.; Pearce, O.; Gamblin, T.; Taufer, M.; Isaacs, K.E.; Bhatele, A. Usability and Performance Improvements in Hatchet. In Proceedings of the IEEE/ACM International Workshop on HPC User Support Tools (HUST) and Workshop on Programming and Performance Visualization Tools (ProTools), Atlanta, GA, USA, 18 November 2020; pp. 49–58. [CrossRef]

Silva, A.B.N.; Cunha, D.A.M.; Silva, V.R.G.; Furtunato, A.F.A.; Souza, S.X.-d.-S. PaScal Viewer: A Tool for the Visualization of Parallel Scalability Trends. In Proceedings of the ESPT/VPA@SC, Dallas, TX, USA, 11–16 November 2018.

Eriksson, J.; Ojeda-may, P.; Ponweiser, T.; Steinreiter, T. *Profiling and Tracing Tools for Performance Analysis of Large Scale Applications*; PRACE—Partnership for Advanced Computing in Europe: Brussels, Belgium, 2016; pp. 1–30. Available online: `https://prace-ri.eu/wp-content/uploads/WP237.pdf` (accessed on 12 January 2020).

Roberts, S.I.; Wright, S.A.; Fahmy, S.A.; Jarvis, S.A. Metrics for Energy-Aware Software Optimisation. In *High Performance Computing, Proceedings of the 32nd International Conference, ISC High Performance 2017, Frankfurt, Germany, 18–22 June 2017*; Kunkel, J.M., Yokota, R.,

Balaji, P., Keyes, D., Eds.; Springer International Publishing: Cham, Switzerland, 2017; pp. 413–430.

Eastep, J.; Sylvester, S.; Cantalupo, C.; Geltz, B.; Ardanaz, F.; Al-Rawi, A.; Livingston, K.; Keceli, F.; Maiterth, M.; Jana, S. Global Extensible Open Power Manager: A Vehicle for HPC Community Collaboration on Co-Designed Energy Management Solutions. In *High Performance Computing, Proceedings of the 32nd International Conference, ISC High Performance 2017, Frankfurt, Germany, 18–22 June 2017*; Kunkel, J.M., Yokota, R., Balaji, P., Keyes, D., Eds.; Springer International Publishing: Cham, Switzerland, 2017; pp. 394–412.

Hackenberg, D.; Ilsche, T.; Schuchart, J.; Schöne, R.; Nagel, W.E.; Simon, M.; Georgiou, Y. HDEEM: High Definition Energy Efficiency Monitoring. In Proceedings of the Energy Efficient Supercomputing Workshop, New Orleans, LA, USA, 16 November 2014; pp. 1–10. [CrossRef]

Roberts, S.I.; Wright, S.A.; Fahmy, S.A.; Jarvis, S.A. The Power-Optimised Software Envelope. *ACM Trans. Archit. Code Optim.* **2019**, *16*, 1–27. [CrossRef]

Boehme, D.; Gamblin, T.; Beckingsale, D.; Bremer, P.T.; Gimenez, A.; LeGendre, M.; Pearce, O.; Schulz, M. Caliper: Performance Introspection for HPC Software Stacks. In Proceedings of the SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Salt Lake City, UT, USA, 13–18 November 2016; pp. 550–560. [CrossRef]

Corporation, I. Intel VTune. Available online: `https://software.intel.com/vtune` (accessed on 15 February 2020).

Pantazopoulos, K.N.; Houstis, E. Performance Analysis and Visualization Tools for Parallel Computing. 1997. Available online: `https://docs.lib.purdue.edu/cstech/1346` (accessed on 20 May 2020).

Gerndt, M.; Ott, M. Automatic Performance Analysis with Periscope. *Concurr. Comput. Pract. Exp.* **2010**, *22*, 736–748. [CrossRef]

Labarta, J.; Gimenez, J.; Martínez, E.; González, P.; Servat, H.; Llort, G.; Aguilar, X. Scalability of Tracing and Visualization Tools. In

Proceedings of the International Conference ParCo, Prague, Czech Republic, 10–13 September 2005; pp. 869–876.

Bienia, C.; Kumar, S.; Singh, J.P.; Li, K. The PARSEC benchmark suite: Characterization and architectural implications. In Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, Toronto, ON, Canada, 25–29 October 2008; pp. 72–81. [CrossRef]

Furtunato, A.F.A.; Georgiou, K.; Eder, K.; Xavier-De-Souza, S. When Parallel Speedups Hit the Memory Wall. *IEEE Access* **2020**, *8*, 79225–79238. [CrossRef]

Silva, V.R.G.; Valderrama, C.; Manneback, P.; Xavier-de-Souza, S. Analytical Energy Model Parametrized by Workload, Clock Frequency and Number of Active Cores for Share-Memory High-Performance Computing Applications. *Energies* **2022**, *15*, 1213. [CrossRef]

A. C. de Melo, "The New Linux 'perf' Tools," *Linux Kongress*, 2010.

Abella-Gonzalez, M., Carollo-Fernandez, P., Pouchet, L., Rastello, F. & Rodriguez, G. PolyBench/Python: Benchmarking Python Environments with Polyhedral Optimizations. *Proceedings Of The 30th ACM SIGPLAN International Conference On Compiler Construction*. pp. 59-70 (2021)

Nikolaev, R. & Back, G. Perfctr-Xen: A Framework for Performance Counter Virtualization. *Proceedings Of The 7th ACM SIGPLAN/SIGOPS International Conference On Virtual Execution Environments*. pp. 15-26 (2011)

Iea, "Data centres and data transmission networks – analysis," Nov 2021. [Online]. Available: `https://www.iea.org/reports/data-centres-and-data-transmission-networks`

J. M. Cardoso, J. G. F. Coutinho, and P. C. Diniz, "Chapter 2 - high-performance embedded computing," in *Embedded Computing for High Performance*, J. M. Cardoso, J. G. F. Coutinho, and P. C. Diniz, Eds. Boston: Morgan Kaufmann, 2017, pp. 17–

56. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780128041895000028

S. Irani, S. Shukla, and R. Gupta, "Algorithms for power savings," *ACM Transactions on Algorithms*, vol. 3, p. 41, 11 2007. [Online]. Available: https://dl.acm.org/doi/10.1145/1290672.1290678

C. Poellabauer, L. Singleton, and K. Schwan, "Feedback-based dynamic voltage and frequency scaling for memory-bound real-time applications," 2005, pp. 234–243.

S. Saha and B. Ravindran, "An experimental evaluation of real-time dvfs scheduling algorithms." ACM Press, 2012, pp. 1–12. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2367589.2367604

I. Pietri and R. Sakellariou, "Energy-aware workflow scheduling using frequency scaling." IEEE, 9 2014, pp. 104–113. [Online]. Available: http://ieeexplore.ieee.org/document/7103444/

L. Mashayekhy, M. M. Nejad, D. Grosu, D. Lu, and W. Shi, "Energy-aware scheduling of mapreduce jobs." Institute of Electrical and Electronics Engineers Inc., 9 2014, pp. 32–39.

M. H. N. Yousefi and M. Goudarzi, "A task-based greedy scheduling algorithm for minimizing energy of mapreduce jobs," *Journal of Grid Computing*, vol. 16, pp. 535–551, 12 2018.

C. Kessler, S. Litzinger, and J. Keller, "Crown-scheduling of sets of parallelizable tasks for robustness and energy-elasticity on many-core systems with discrete dynamic voltage and frequency scaling," *Journal of Systems Architecture*, vol. 115, p. 101999, 5 2021. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1383762121000175

M. S. Ajmal, Z. Iqbal, F. Z. Khan, M. Bilal, and R. M. Mehmood, "Cost-based energy efficient scheduling technique for dynamic voltage and frequency scaling system in cloud computing," *Sustainable Energy Technologies and Assessments*, vol. 45, p. 101210, 6 2021. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2213138821002204

P. Agrawal and S. Rao, "Energy-efficient scheduling: classification, bounds, and algorithms," *Sādhanā*, vol. 46, p. 46, 12 2021. [Online]. Available: http://link.springer.com/10.1007/s12046-021-01564-w

V. R. G. da Silva, A. B. N. da Silva, C. Valderrama, P. Manneback, and S. Xavier-de Souza, "A minimally intrusive approach for automatic assessment of parallel performance scalability of shared-memory hpc applications," *Electronics*, vol. 11, no. 5, 2022. [Online]. Available: https://www.mdpi.com/2079-9292/11/5/689

TOP500, "TOP500 Supercomputer Sites," [Online]. Available: https://www.top500.org.

Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, Gaël Varoquaux, *API design for machine learning software: experiences from the scikit-learn project*, in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122.

H. Drucker, C. J. C. Burges, L. Kaufman, A. Smola, and V. Vapnik, *Support vector regression machines*, *Advances in Neural Information Processing Systems*, vol. 1, pp. 155–161, 1997. doi:10.1.1.10.4845. http://papers.nips.cc/paper/1238-support-vector-regression-machines.pdf

R. Bollapragada, D. Mudigere, J. Nocedal, H.-J. M. Shi, and P. T. P. Tang, *A Progressive Batching L-BFGS Method for Machine Learning*, 2018. arXiv:1802.05374.

T. Tieleman and G. Hinton, *Lecture 6.5 - RMSProp: Divide the gradient by a running average of its recent magnitude*, *COURSERA: Neural Networks for Machine Learning*, 2012. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

L. Bottou, *Large-Scale Machine Learning with Stochastic Gradient Descent*, *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT 2010)*, pp. 177–186, 2010. http://leon.bottou.org/publications/1001

D. P. Kingma and J. B. Adam, *Adam: A Method for Stochastic Optimization*, *arXiv preprint arXiv:1412.6980*, 2014. https://arxiv.org/abs/1412.6980

Energy-optimal configurations for High-Performance Computing applications: automated low-impact characterization and performance optimization of shared-memory applications

Energy consumption is key to enabling exascale High-performance Computing (HPC). However, energy-optimized hardware and software combinations could still be inefficient if the software operates poorly.

This work proposes a set of tools, models, and algorithms for energy optimization aimed at high-performance computing based on knowledge of the application and the specific hardware architecture. The main contributions of this work are.

A framework called Parallel Scalability Suite (PaScal Suite) automatically measures and compares multiple executions of a parallel application according to various scenarios characterized by input size, number of threads, cores, and frequencies. As a result, PascalSuite can automate designing application models with an overhead of less than 1%.

An entire system energy model based on the CPU frequency and the number of cores. The model aims to understand and optimize the energy behavior of parallel applications in HPC systems according to application parameters, such as the degree of parallelism, input load, and CPU parameters related to dynamic and static power.

A methodology that combines measurement data with a heuristic algorithm to provide insights into choosing the best phase divisions. Our heuristic can reduce the scan space from 10^{7000} to 10^2 with an average error of 10% and up to 38% reduction in energy consumption using optimal distribution compared to standard Linux DVFS.

A novel normalized time representation of the application characterizes the application parameters and model, named application fingerprint.

UMONS
Université de Mons