OPTKIT: Unleash the Green Potential of Your CPU Code

by

Osman Yasal

A Dissertation Submitted to the
Graduate School of Sciences and Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in

Computer Science and Engineering

KOÇ ÜNİVERSİTESİ

August 2, 2024

OPTKIT: Unleash the Green Potential of Your CPU Code

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

Osman Yasal

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

_____

Assoc. Prof. Didem Unat (Advisor, Koç University)

_____

Prof. Öznur Özkasap (Koç University)

_____

Prof. Tolga Ovatman (Istanbul Technical University)

Date: _____

ABSTRACT

OPTKIT: Unleash the Green Potential of Your CPU Code
Osman Yasal
Master of Science in Computer Science and Engineering
August 2, 2024

The analysis of energy consumption in software has traditionally taken a back seat to performance analysis. However, increasing environmental concerns, power and cooling limitations in expanding HPC systems and data centers, and the need for energy efficiency in power-constrained devices such as mobile phones, tablets, laptops, and IoT devices have brought significant focus to this area. While performance analysis has a wide range of tools, those specifically designed for software energy efficiency are scarce. This study introduces the Optimizer Toolkit (OPTKIT), a dedicated library and toolset designed for the energy analysis and optimization of software runtime. OPTKIT utilizes perf_event_open kernel call for PMU and RAPL, and msr-safe for cpu frequency operations. Inside the tool, there are *Query** classes where the user can query anything regarding cpu, energy, and performance before taking any action. It includes useful utility tools that help users streamline the optimization process.

In this work, we leverage OPTKIT to evaluate the potential for reducing a system's energy consumption by strong-scaling core numbers in a given application. We then use these data to further reduce energy consumption through co-scheduling applications. In another use-case, we investigate the impact of CPU frequency on energy consumption and develop a methodology to dynamically detect the optimal frequency at runtime and adjusts frequencies based on execution phases to maximize energy savings.

We find that not all applications scale linearly, and allocating appropriate resources yields better energy savings. Co-scheduling applications by using unassigned resources often saves more energy than serial execution counterparts. Although manually finding the best frequency offers the highest energy savings, our automated approach achieves up to 31.83% energy savings compared to normal execution of applications. The OPTKIT tool is publicly available on [34].

ÖZETÇE

OPTKIT: CPU Kodunuzun Yeşil Potansiyelini Ortaya Çıkarın
Osman Yasal
Bilgisayar Bilimleri ve Mühendisliği, Yüksek Lisans
2 Ağustos 2024

Yazılımda enerji tüketiminin analizi, geleneksel olarak performans analizinin gerisinde kalmıştır. Ancak, artan çevresel farkındalık, gelişen yüksek performanslı systemler ve veri merkezleri, güç sınırlı, iot, mobil, dizi üstü bilgisayarlar vb. cihazlardaki enerji tüketiminin optimize edilmesi gibi sebepler bu alana olan ilgiyi arttırmıştır. Performans analizi geniş bir araç yelpazesi sunarken, yazılım enerji verimliliği için özel olarak tasarlanmış araçlar sınırlıdır hatta pek fazla yoktur. Bu çalışma, yazılım çalışma zamanının enerji analizi ve optimizasyonu için tasarlanmış özel bir kütüphane ve araç seti olan Optimizer Toolkit (OPTKIT[34])'i tanıtmaktadır.

OPTKIT, PMU ve RAPL için perf_event_open çekirdek çağrısını ve CPU frekans işlemleri için msr-safe'i kullanır. Araç içinde, kullanıcıların herhangi bir eylemde bulunmadan önce cpu, enerji ve performans ile ilgili her şeyi sorgulayabileceği *Query** sınıfları bulunmaktadır. OPTKIT kullanıcıların optimizasyon sürecini kolaylaştırmalarına yardımcı olan faydalı ekstra araçlarla birlikte gelir.

Bu çalışmada, OPTKIT kullanarak bir sistemin enerji tüketimini değerlendirmeyi ve belirli bir uygulamada çekirdek sayısını artırarak veya azaltarak bu tüketimi azaltma potansiyelini araştırıyoruz. Daha sonra, bu verileri uygulamalar eş zamanl çalştrarak enerji tüketimini daha da minimize etmek amacyla kullanyoruz. Alternatif bir senaryoda ise, CPU frekansının enerji tüketimi üzerindeki etkilerini analiz ediyoruz ve çalışma süresinde optimal frekansı dinamik olarak belirleyip enerji tasarrufunu maksimize etmek için frekans ayarlama yöntemleri geliştiriyoruz.

Tüm uygulamaların doğrusal olarak ölçeklenmediğini ve uygun kaynak tahsisinin daha iyi enerji tasarrufu sağladığını buluyoruz. Atanmamış kaynakları kullanarak uygulamaları birlikte zamanlamak, genellikle seri yürütme eşdeğerlerinden daha fazla enerji tasarrufu sağlar. En iyi frekansı manuel olarak bulmak en yüksek enerji tasarrufunu sağlasa da, otomatik yaklaşımımız normal uygulama yürütmesine kıyasla %31.83'e kadar enerji tasarrufu sağlamaktadır.

TABLE OF CONTENTS

List of Figures **vii**

Abbreviations **ix**

Chapter 1: Introduction **1**

Chapter 2: Related Work **4**

2.1 Related Work 4

Chapter 3: Background & Motivation **9**

3.1 Performance Monitoring Units(PMUs) 9

3.2 Running Average Power Limit (RAPL) 10

3.3 CPU Frequency 11

3.3.1 Frequency Governors 12

Chapter 4: Optimizer Toolkit **14**

4.1 Overview 14

4.2 Installation and Using 21

4.3 Measurement Verification 22

4.4 Implementation Details 22

Chapter 5: Tool Demonstration **28**

5.0.1 Energy-Aware Strong Scaling and Co-Scheduling 29

5.0.2 Dynamic Frequency Scaling Effects 34

5.0.3 Custom Frequency Governor 36

5.0.4 Cross Platform Execution Results 41

Chapter 6: Conclusion **50**

Chapter 7: Future Work **51**

Bibliography **52**

LIST OF FIGURES

4.1 PMU Macro Definition 18
4.2 A100 RAPL Sleep MERIC Result 22
4.3 A100 RAPL Sleep OPTKIT Result 23
4.4 Example Case Program Folders 25
4.5 Example Case Enriched RAPL 26
4.6 Example Case Intensity Output 26
4.7 Example Case RAPL Output 27

5.1 Strong Core Scaling Methodology 29
5.2 Hotspot3D Strong Scaling 30
5.3 Hotspot3D Suggestion for Co Scheduling 30
5.4 Gemm Strong Scaling (cores 1&2 clipped out for readability) 31
5.5 Gemm Suggestion for Co Scheduling (cores 1&2 clipped out for readability) 32
5.6 Heartwall Strong Scaling (cores 1&2 clipped out for readability) . . . 33
5.7 Heartwall Suggestion for Co Scheduling (cores 1&2 clipped out for readability) 34
5.8 LavaMD Strong Scaling (cores 1&2 clipped out for readability) 35
5.9 LavaMD Suggestion for Co Scheduling (cores 1&2 clipped out for readability) 36
5.10 Strong Core Scaling Detailed Methodology 37
5.11 Co Scheduling Methodology 38
5.12 Single Execution Results for Detailed Methodology 39
5.13 Sequential Execution Results for Detailed Methodology 40
5.14 Co-Scheduled Execution Results for Detailed Methodology 41

5.15 Heat-Map Gemm 42
5.16 Overview of Custom Governor and its Parameters 44
5.17 Frequency Scaling Results of Model Benchmarks 45
5.18 A100 Machine Gemm Core Scaling 47
5.19 A100 Machine LavaMD Core Scaling 47
5.20 A100 Machine Heartwall Core Scaling 48
5.21 A100 Machine Hotspot3D Core Scaling 48

ABBREVIATIONS

OPTKIT	Optimiser Toolkit
TPC	Thread per Core
SMT	Simultaneous Multithreading
OpenMP	Open Multi-Processing
CPU	Central Processing Unit
EDP	Energy Delay Product
K-EDP	Kilo Energy Display Product
CI	Computational Intensity
MI	Memory Intensity
Freq	Frequency
TDP	Thermal Design Power
MEM	Memory
ARCH	Architecture
OS	Operating System
PP0	Power Plane 0
PP1	Power Plane 1

Chapter 1

INTRODUCTION

With the emerge of the HPC & Cloud Systems, providers mainly employed the "pay-as-you-go" model for their pricing policy[11]. While this model traditionally centered on the utilization of system resources and the execution time, there is now a trend towards incorporating power consumption into their pricing strategies [7] [30] [38] [18] [6].

In addition to pricing concerns, another challenge in HPC systems related to energy emissions and power consumption is the expansion of these systems through the addition of new nodes. Existing cooling systems, which are already handling significant power consumption and energy emissions, face increased inefficiencies as new nodes are integrated. The current cooling infrastructure is beginning to struggle with the growing demands imposed by these additional nodes [41].

The perspective on performance and energy consumption has often been seen as "the glass is half empty." The general assumption is that improving software performance leads to reduced energy consumption because faster task completion on CPUs, GPUs, or other computing units allows them to enter an idle (low power) state, resulting in less energy consumption over a given time T compared to when they are not active. However, this is not always the case. It is possible to achieve higher energy savings by maintaining the same performance or even slightly compromising the performance [42]. Various factors, such as the efficiency of the hardware and the nature of the workload, can influence the relationship between performance and energy consumption.

Given the reasons, application developers aim to improve execution time to save money and provide faster solutions, while system providers expect existing applica-

tions to emit less energy, allowing for the expansion of their HPC systems. Consequently, numerous performance tools have emerged over the years, offering solutions ranging from specific optimizations to broader performance analysis. Although a few of these performance tools measure energy consumption, their primary focus remains on improving performance rather than reducing energy and power consumption. Thus, the idea of improving energy savings is mostly an uncharted territory for developers.

Within the current research landscape, there are only few tools, such as *PMT [14] and* MERIC *[42]*, that focus exclusively on the analysis of energy consumption. Although PMT effectively measures energy consumption across devices such as CPUs, GPUs, and FPGAs, it does not offer solutions to reduce energy consumption. Conversely, MERIC addresses CPU frequency adjustments for energy savings, but this requires users to analyze their code blocks to determine the appropriate frequency, which can significantly burden users with additional workload and none of these energy tools covers performance insights for the code blocks, forcing developers to search and learn new tools for performance analysis.

OPTKIT fills the existing gap by providing an extensive API that supports both *performance insights* from low level monitoring (PMU counters) to high level use cases (computation intensity, memory and cache intensities, bad speculation, branch misses) and *energy insights* (energy readings, frequency adjustments, runtime optimisations) along with utility and graphical tools. While our tool shares similarities with the *Meric* tool in adjusting frequency for a specified code block for energy savings, we offer this capability on runtime by eliminating the need for detailed analysis and any redundant run of the target application.

In conclusion our contributions can be listed as follows, we offer a tool that facilitates both energy and performance analysis during the software development phase of an application, enabling developers to design and optimize their codebase more efficiently. Additionally, it can be applied to existing applications to reduce energy consumption in run-time through its energy-focused custom frequency governor. Finally, OPTKIT comes with easy-to-use cli tools allowing users to perform energy analysis for various scenarios and optimisations effortlessly.

The thesis is organized as follows: Chapter 2 reviews relevant studies and the need for OPTKIT; Chapter 3 provides foundational information for its development; Chapter 4 covers its overview, installation, and implementation details; Chapter 5 demonstrates its features; and Chapters 6 and 7 present conclusions and future work.

Chapter 2

RELATED WORK

2.1 Related Work

As described in the *Introduction* section there are many tools for software performance analysis, yet only a few tools for energy consumption. In this section, we discuss existing performance tools and put other works regarding energy consumption. Comparison of our tool with other eminent tools can be found in the Table 2.1.Before diving into the specifics of related work, it is important to clarify the meanings of certain terms. When we refer to *in development support*, we mean that the tool/library can provide feedback during the development phase of an application and shows how recent updates affect your current state. In our tool OPTKIT, you can get the performance and energy consumption metrics for a certain block of code, allowing you to improve its performance as you program. On the other hand, when we mention runtime optimizations, we mean that the tool/library optimizes its performance as the software is being run. Our tool OPTKIT also supports this by dynamically changing the core-uncore frequency at runtime to optimize energy consumption.

Intel's Vtune and Advisor [23], along with AMD's uProf [9], stand out among the performance tools available. Additionally, tools like *perf* originate from the Linux environment, while others such as *PAPI, Score-P, ExtraP, LIKWID, valgrind and others* [37] are developed by third-party contributors.

Each tool carries its own set of advantages and drawbacks. Hardware-specific tools like those from Intel and AMD provide generic analysis such as hotspot,tmam, gpu, memory utilisation, but are limited to their particular CPUs. Conversely, *perf* is hardware independent but focuses solely on postmortem analysis with some predefined events (e.g., cache hits/misses, page faults, branch misses) and specific

PMC event monitoring, but lacks in-development support, runtime optimisations, and graphical support like results graphs.

PAPI offers a robust API and in-development support for performance counters across various hardware, however lacks in high-level use cases, graphical support, and runtime optimisations.

The above-mentioned tools can be considered as "generic purpose". Tools such as callgrind [12] (cache simulation), Archer [10](data race detector for OMP programs) and ComDetective [5] (communication detection for threads) are considered for specific reasons. Comprehensive overview of additional performance tools and their objectives can be seen at this flyer [37].

It is worth mentioning that none of these performance tools specifically addresses energy consumption. Although some tools are capable of reading energy consumption data (like PAPI[36] and uProf[9]), they lack the ability to attribute energy consumption to code blocks, runtime optimizations to reduce energy consumption, and providing tools that ease developers' tasks in this regard.

Authors in [19] provides an initial guide on utilizing RAPL for assessing the energy consumption of software code blocks. The frequency at which RAPL offers measurements is discussed, and one of the accurate approaches to accurately interpreting the values is outlined. The work in [26] serves as a guide on Intel's RAPL for measuring power, covering both its benefits and drawbacks. Experiments demonstrate the accuracy of RAPL with minimal performance impact. Noteworthy challenges, such as driver support, are identified, along with proposed solutions for specific scenarios. The authors in [25] investigate efficient power profiling for advanced schedulers, focusing on core scaling to reduce peak power. The proposed approach utilizes short profiling runs for instantaneous power traces, significantly decreasing the profiling duration while maintaining accuracy. The work in [35] presents a survey aimed at gauging programmers' knowledge regarding the energy consumption of software. Understanding the general opinion among programmers regarding energy-efficient software is crucial for the development of improved tools. Work in [8] investigates the accuracy of Running Average Power Limit (RAPL) for measuring memory power consumption on heterogeneous high-performance comput-

ing systems, finding that RAPL overestimates power usage and shows inconsistencies, particularly at lower memory loads and across multiple sockets. Study in [15] proposes a new power measurement and limiting architecture for main memory in enterprise servers, demonstrating up to 40% lower performance impact compared to existing methods, thereby improving efficiency in power capping and cooling. Authors in [43] presents a dynamic application-aware power scheduling (DAPS) strategy for power-capped large-scale HPC clusters, achieving up to 17% better performance compared to static power distribution by dynamically adjusting power limits based on hardware usage and application sensitivity. The work in [27] introduces an energy profiling module for IgProf, using the RAPL interface to measure energy consumption in scientific computing workloads, and shows promising results with a close correlation between execution time and energy usage in single-threaded programs. Work in [33] presents a detailed analysis of OpenMP programs for power and performance using the OpenMP Runtime API (ORA) and RAPL sensors, revealing that optimal execution performance does not always correlate with the best energy usage, and identifying waiting times and queue delays as major contributors to high power consumption. Authors in [22] analyzes power management in Intel Haswell processors, used in the Trinity supercomputer, revealing that real-time power monitoring can increase power consumption, while optimizing P-states, enabling hyperthreading, and employing differentiated core affinity strategies can significantly enhance energy efficiency and performance. Authors in [17] proposes a two-level hierarchical power management approach for exascale supercomputers, combining PPartition for macro-level power distribution and PTune for micro-level optimization, achieving up to 29% performance improvement and freeing 40% more processors compared to naive power allocation, with overall throughput gains of 5-35%. Authors in [24] develops a model to predict CPU power consumption based on task-specific processor cycles, aiming to optimize energy efficiency by scheduling tasks to minimize idle core energy use and avoid unnecessary wakeups, based on observed power profiles of different tasks. The work [16] examines the effects of thread and frequency scaling on performance and energy consumption in modern multi-cores, finding that optimal performance occurs with 8-12 threads and maximum

frequency, while minimal energy consumption is achieved with a frequency between 2.0 and 2.4 GHz, with the best balance of performance and energy efficiency at 8-12 threads and 2.8-3.1 GHz. Work in [40] discusses Intel's Alder Lake architecture, highlighting its approach to managing performance and energy efficiency through heterogeneous core designs, frequency scaling, idle states, integrated energy measurement, and processor feedback mechanisms. Readex project [13] is developing a tool suite that creates a tuning model for HPC applications by analyzing configurations during design-time, which are then dynamically adjusted at runtime to enhance energy efficiency. Authors in [21] evaluates the impact of dynamic voltage frequency scaling (DVFS) on performance, energy efficiency, and their product (PxEE) using SPEC CPU2017 benchmarks on an Intel Core i7 processor. It introduces four DVFS-based techniques driven by performance metrics from the processor's monitoring unit, demonstrating significant improvements in energy efficiency (EE) and PxEE—up to 92% and 48%, respectively—compared to existing methods, with particularly notable gains for memory-intensive benchmarks. Work in [28] proposes an energy-centric dynamic voltage frequency scaling (eDVFS) method to minimize total energy consumption in servers by adjusting CPU frequency based on shared resource contention, demonstrating greater energy efficiency and speed compared to the standard "on-demand" CPU governor. Work in [20] evaluates the energy impact of co-scheduling parallel workloads on multi-core processors and introduces a novel multi-level technique that improves energy efficiency by up to 22% compared to existing methods. Work in [29] shows how extending a user-space run-time resource manager to utilize performance counters can optimize both execution time and energy consumption for mixed workloads on multicore architectures, achieving a 49.9% improvement in energy-delay-product (EDP) compared to standard Linux and a 13.4% improvement over previous methods.

The work in [17] addressing exascale power constraints highlights that uniformly distributing power across processors leads to sub-optimal performance due to processor variation; their proposed 2-level hierarchical variation-aware approach, with PPartition for macro-level power allocation and PTune for micro-level job-centric optimization, achieves up to 29% performance improvement and 5-35% throughput

enhancement compared to naive power distribution methods.

Work in [31] introduces a novel framework for enhancing energy efficiency in IoT application development by combining static analysis with dynamic instrumentation, providing accurate energy consumption estimates and optimization insights, and demonstrating significant energy reductions in industrial use-cases.

PMT[14] is a library utilising RAPL to measure energy consumption of code blocks both for C/C++ and Python environments. Additionally it extends its functionality to include energy reading from NVIDIA and AMD GPUs and FPGAs. Despite its potential, the tool currently does not support any insight regarding performance, lacks maturity in presenting meaningful charts and practical use-cases and also doesn't include any runtime optimisation support.

Meric and Radar Generator [42] is another library/tool-set that can be used to manually instrument and analyze C/C++ and FORTRAN applications along with NVIDIA GPUs through NVML in terms of energy consumption. In addition to energy readings, MERIC can modify environmental and hardware parameters while the application is running, resulting in energy savings. MERIC solely focuses on energy consumption as they state does not provide any utilities regarding performance analysis.

Tool	PMU Reads	RAPL Reads	Runtime Optimisation	Built-in Visualisation	PMU Insights	RAPL Insights	Cross CPU
OPTKIT	✓	✓	✓	✓	✓	✓	✓
PAPI	✓	✓					✓
MERIC		✓	✓	✓		✓	✓
Vtune	✓			✓	✓		
uProf	✓	✓		✓	✓		
PMT		✓		✓			✓

Table 2.1: Energy Tool Comparisons

Chapter 3

BACKGROUND & MOTIVATION

Within the developer community, the primary goal during software development is often focused on making the application faster. This emphasis on speed can make it initially unclear to developers why they should prioritize or consider energy efficiency in their projects. In the related survey [35], some developers expressed the belief that ensuring energy efficiency is the responsibility of hardware manufacturers.

While this statement holds validity, it is fading away, akin to the era when programmers used to anticipate new hardware releases for performance improvements. That's why it's now developers' job to look for energy efficiency.

Fortunately, hardware vendors have added some useful components, namely performance counters and RAPL, to their contemporary hardware to be used for developing tools and libraries in this purpose.

3.1 Performance Monitoring Units(PMUs)

Performance monitoring unit (PMU) is a hardware component that is integrated into contemporary CPUs to collect various performance metrics during the execution of applications.

The primary purpose of a PMU is to provide insight into the performance characteristics of a system, helping developers, researchers, and system administrators optimize software and improve overall system efficiency.

PMU architecture provides some counters that can be configured by programmers to count specific PMU events. The number of performance counters, their configuration, accessing the counters, and the available events are entirely depends on the hardware provider.

Currently, prominent hardware providers such as Intel, AMD, and ARM offer

PMU architecture with a multitude of events such as *Instructions Retired, Cache Misses, Branch Mispredictions, L1/L2/L3 Cache Accesses, Floating-Point Operations, Pipeline Stalls...* Nevertheless, these events exhibit variations not only across different vendors but also within distinct micro-architectures of the same vendor. This is why discovering a universal solution that functions effectively across all CPUs remains a somewhat laborious task, yet it is advantageous to pursue. In depth analysis and comparisons of these PMU units can be found on this work [39].

*3.2 Running Average Power Limit (*RAPL*)*

RAPL stands for *Running Average Power Limit*, and it is a feature in modern processors that allows for monitoring and controlling the power consumption dynamically. RAPL is primarily used in Intel processors and is part of Intel's broader set of power management technologies. However, it's also implemented in AMD processors.

In-depth investigation, as highlighted in [] demonstrates that RAPL is more precise than plug-wise power meters, in terms of higher reporting frequency and attribution of energy consumption. While power meters demonstrate good accuracy, they are limited to measuring power once per second. In contrast, RAPL can update every millisecond at a frequency of 1000Hz. Additionally, power meters lack the capability for detailed analysis, whereas RAPL enables the measurement of power consumption across various domains.

The power domains reported by RAPL are given in the Table 3.1. However, it is important to emphasize that not all processors equipped with RAPL can provide all of this information. For example dram support only exists in server processors, AMD chips mostly report package consumption and dram(based on if it's server) on the flip side Intel chips tend to give PP0,PP1 as well.

RAPL offers data on a per-socket basis, unlike PMUs, RAPL does not posses the capability to distinguish measurements for individual applications. Therefore, values obtained from a socket reflects the collective impact of all applications running on the aforementioned socket at a given time. The limitation of RAPL in

PowerPlane0	All cores in the socket
PowerPlane1	Offcore device (integrated graphics...)
Package	Socket energy consumption
Dram	Dram energy consumption
Psys	Package + eDRAM + PCH...

Table 3.1: RAPL Domains

distinguishing measurements for individual applications poses a challenge when it comes to monitoring energy consumption and attributing to code blocks, especially when compared to the event monitoring capabilities of code blocks with PMUs. To address this issue, one approach could be to measure the energy consumption of the idle system and then subtract that measurements from the one obtained when measuring idle system plus our program running. The other approach, which is more accurate and employed in the use cases of this paper, is to have a two socket system and separate one socket only for our program while other socket is running the system.

Based on the available information as of the time of writing this paper, there are 3 ways of reading RAPL measurements in Linux which are; reading via power-cap interface */sys/class/powercap/intel-rapl/* that is available since Linux 3.13, using perf-event interface with Linux 3.14 or newer or using raw-access to MSR registers under */dev/msr*. It should be noted that powercap and msr methods require root access; on the other hand, the perf-event interface does not require root access for readings, as long as $perf\_event\_paranoid < 0$, one can read perf-events with no additional credentials.

3.3 CPU Frequency

CPU frequency becomes significant when considering energy consumption; core frequency and uncore frequency, each playing a role in power management.

Core frequency is the frequency at witch CPU cores themselves operate. It

determines the speed at which instructions are processed.

Uncore frequency is the frequency at which components outside of the CPU cores operate, including the memory controller, caches, PCIe controller, and other integrated components.

In order to modify the CPU frequency according to user needs, it's necessary to switch the current CPU driver to *acpi-cpufreq* from */etc/default/groub*. This can be done by turning off intel_pstate=disable and amd_pstate=disable for Intel and AMD processors respectively, and setting acpi-cpufreq=enabled then grub update by the "update-grub" command and rebooting.

After these steps, the core frequency and its associated settings can be modified within the */sys/devices/system/cpu/cpu*/ cpufreq/* directory. However, adjusting the uncore frequency is more complex than adjusting core frequencies since it involves manipulating the MSR (Model Specific Register) settings.

Our tool OPTKIT, makes direct modifications to the referred files to adjust the CPU core frequency, while utilizing the msr-safe library [32] to adjust the uncore frequency.

Additionally, it is worth noting that frequency adjustments are nearly trivial operations, typically taking an average of 1-3ms in our measurements. Although these files can be read by any user altering these files requires a *sudo* privilege. From the system admin perspective, it may be feasible to create a special user-group, that has the privilege to change the content of these files, and run the programs with that user-group.

3.3.1 Frequency Governors

Frequency governors are part of the CPU frequency scaling subsystem. They control the CPU's frequency (speed) to balance performance and power consumption based on system load. The governors adjust the CPU frequency dynamically to optimize for either power efficiency or performance. Available frequency governors in contemporary Linux systems are given in the Table 3.2. MERIC [42] mention that it is possible to find a frequency combination that has more energy savings

indirectly proving that current frequency drivers are inadequate for energy savings. Our OPTKIT library includes a custom governor that is automatically inherited by any application using OPTKIT. In one of our use cases, we demonstrate the efficiency of this custom governor, which optimally finds the best frequency combination according to the workload.

Governor	**Description**
performance	Run the CPU at the maximum frequency, obtained from `/sys/devices/system/cpu/cpuX/cpufreq/scaling_max_freq`.
powersave	Run the CPU at the minimum frequency, obtained from `/sys/devices/system/cpu/cpuX/cpufreq/scaling_min_freq`.
userspace	Run the CPU at user-specified frequencies, configurable via `/sys/devices/system/cpu/cpuX/cpufreq/scaling_setspeed`.
ondemand	Scales the frequency based on current load. Jumps to the highest frequency and backs off when idle time increases.
conservative	Scales the frequency based on current load. Scales the frequency more gradually than ondemand.
schedutil	Scheduler-driven CPU frequency selection.

Table 3.2: CPU Frequency Scaling Governors

Chapter 4

OPTIMIZER TOOLKIT

This chapter introduces the foundation of this work - the *Optimiser Toolkit* and its core components and code samples.

4.1 Overview

OPTKIT is an extensively customisable library/tool-set crafted in C++11 that can easily be used in measuring energy consumption of software blocks and tuning hardware parameters during application runtime for energy efficiency. It's overhead depends on the amount of performance counters and RAPLreads as well as the number of instructed regions.

Besides the other tools mentioned in the *"Related Tools"* section, OPTKIT aims to be integrated into the development process in the same way as any other library. It can be solely utilized for gathering energy and performance metrics like energy/power usage, computation intensity, etc., to conduct in-development stage code analysis and/or future code refactoring or it can be a part of your software since it is able to change the CPU frequency at runtime to improve energy efficiency or due to its simplicity and robustness, it can be used in the development of other performance/energy tools.

OPTKIT is structured based on the RAII principle, commencing monitoring from the point of its definition until the conclusion of the associated scope. Utilizing the perf_event_open system call for both PMU events and RAPLenergy metrics without necessitating *root* privileges for execution.

OPTKIT uses *libpfm4* for PMU related queries, *perf_event_open* linux system call for accessing RAPLcounters & configuring PMU counters, *MSR-SAFE* library to change the CPU uncore frequency, */sys/devices/system/CPU/CPU*/CPUfreq/* di-

beacon_rapl	Emits energy consumption of the CPU in given frequency (1 sec. by default)
beacon_freq	Emits CPU core frequencies for each cores in given frequency (1 sec. by default)
freq	change core-uncore frequency or reset to default values
probe	Lists system specs for RAPL, PMU & CPU Frequency
enrich	Enriches Energy & Pmu reading *JSON reports to power, total energy consumption, edp, etc.
core_scaling	Performs core strong scaling for OpenMP programs by changing the env. variable OMP_NUM_THREADS
co_schedule	Co-schedules programs to specified cores and sockets
freq_scaling	strong scales core-uncore frequencies for a give application
vis_line	Creates line chart based on the result of core_scaling tool
vis_bar	Creates bar chart based on the input file
vis_freq_heatmap	Creates a heatmap by using the results of freq_scaling

Table 4.1: OPTKIT Utilility Tools

rectory for changing the CPU core frequencies and scripting languages *(bash,python3)* for some utility tools. Use-cases heavily utilised OPTKIT's utility programs given in the Table 4.1.

These tools start with "optkit_" prefix. While some of these tools are bash wrappers that use existing Linux command-line tools, all visualization tools and the enrich tool are written in Python3 and matplotlib and others make use of OPTKIT itself. Upon installation of OPTKIT, these tools are compiled and installed in "/usr/local/bin" so that they can be accessible in the Linux terminal. Here core_scaling and co_scheduling are bash wrappers; The core_scaling requires the socket number and a command to execute. Then by starting from 1 and 2 it increments the OMP_NUM_THREADS environment variable by 2 up to the maximum core number available in the selected socket then executes the command specified. When

no sockets provided, it uses all available sockets on the system. On the other hand the co_scheduling uses Linux's "taskset" command for co_scheduling. Users should set the core numbers and applications as parameters. Then the taskset command is executed for each application by assigning them to the specified cores. Upon termination of all tasks, the command prints out an information, upon canceling the utility, a kill signal is sent to applications given. The freq_scaling tool executes the given program for each core-uncore frequency pairs, it uses the probe tool to retrieve frequency limits and sets *OPTKIT_SOCKETXX__ENABLED and OPTKIT_SOCKETXX__CORE/UNCORE_FREQ* enviroment variables then upon initialisation OPTKIT automatically looks for these enviroment variables, if exists it sets the core-uncore frequency to corresponding CPU then resumes execution of the program. All visualisation tools having "vis" in the name are developed with python3 and the matplotlib library. These visualisation tools requires you to specify either number of files or a location to read files. Input files must be the output of the "enrich" tool. Enrich tool is also written in python3, OPTKIT dumps minimum amount of information as JSON files to reduce the overhead, enrich tool is used to generate other matrics that are useful to the user, similar to other tools it also takes either a file or a directory. Once executed it generates JSON files with the postfix of "*__enriched.JSON". All other tools, namely *beacon_rapl, beacon_freq, freq, probe* are build upon the OPTKIT and written in C++11. Tools *beacon_rapl and beacon_freq* use RaplProfiler and CPUFreq classes to print out energy consumption and current core-uncore frequencies. By default the period is 1 sec. However, this can be set by the user as a parameter. The Freq utility uses OPTKIT's CPUFrequency class to set the core-uncore CPU frequency of a socket. Probe utility uses all *Query*** classes of OPTKIT to print out the fundamental information about the system.

Any feature that OPTKIT provides is carried out by only a few classes and users of the tool can easily extend or add features according to their will. Note that there are numerous monitoring configurations for any of these classes, and users who want to extend or add new features must know what they do. By default, OPTKIT provides C style macros with default configurations which cover most of the real world

cases. OPTKIT's most used building blocks are given in listing[4.1]. All PMU events are located in the "src/core/events" folder within structs. Users can utilize these events with *BlockProfiler and BlockGroupProfiler* classes. The *BlockGroupProfiler* class groups all specified events (up to the maximum number of available PMCs; if exceeded, the tool will notify the user) into a single unit, treating them as a one combined event. This approach is beneficial if you want to ensure that no events are missed. In contrast, the *BlockProfiler* class does not group events. It allows you to set more events than the available PMCs, resulting in event multiplexing. In this case, some events are assigned to PMUs while others are temporarily unassigned and replaced after a certain period. This method is suitable if simultaneous treatment of events is not required. These classes also take a configuration class as their parameter namely *ProfilingConfig*. With this class you can specialise how profilers behave. It contains following parameters *dump_results_to_file, is_reset_after_read, is_grouped, pid, cpu, perf_event_config*. By default those classes dump results upon exiting a block, reset the counter after each read operation and configured to monitor the current program regardless of how many threads it executes. By using the perf_event_config configuration file, one can specialise its profiler as they wish. *RaplProfiler* also has its own configuration class named *RaplConfig* which contains similar parameters with the same default values like *read_method, monitor_domain, is_reset_after_read, dump_results_to_file*. Here rapl_method is the reading method mentioned in the RAPL background section. By default, it is PERF, but it can be MSR or POWERCAP. The other important parameter is the monitor_domain, which corresponds to RAPL domains, by default it is all but one can specialise this to PP0, PP1, dram etc. When it comes to the events that will be passed as parameters to these classes, it is possible to use "events" folder and select related events but for simplicity, "src/core/event_recepies" folder is created in "cpu/pmu_name" format. In these recepies, each PMU has a different name space with the same static class with the name "Recepies". Since the source code is available, one can extend this class as they wish for other measurements. To specialise with macros, you need to put definitions into files with "folder_location/relevancy" or you can write your own. An example for existing profiling_pmu macros is given in the Figure 4.1. One

```
#ifndef OPTIMIZER_TOOLKIT_CORE__SRC__CORE__PROFILING__PMU__PROFILING_PMU_HH
#define OPTIMIZER_TOOLKIT_CORE__SRC__CORE__PROFILING__PMU__PROFILING_PMU_HH

#include <deployment_config.hh>
#include <block_group_profiler.hh>
#include <block_profiler.hh>
#include <libpfm4_wrapper.hh>
#include <pmu_event_manager.hh>
#include <gantt_instrumentor.hh>
#include <query_pmu.hh>

using optkit::core::pmu::BlockGroupProfiler;
using optkit::core::pmu::BlockProfiler;
using optkit::core::pmu::PMUEventManager;
using optkit::core::pmu::QueryPMU;

#if defined(CONF__PMU__MACROS__ENABLED) && CONF__PMU__MACROS__ENABLED == 1
#define OPTKIT_PERFORMANCE_EVENTS(block_name, event_name, variable_name, ...) \
    OPTKIT_CORE_GANTT_PROFILE_SCOPE(block_name);                               \
    BlockProfiler variable_name { block_name, event_name, __VA_ARGS__ }

#define OPTKIT_PERFORMANCE_BLOCK_EVENTS(block_name, event_name, variable_name, ...) \
    OPTKIT_CORE_GANTT_PROFILE_SCOPE(block_name);                                    \
    BlockGroupProfiler variable_name { block_name, event_name, __VA_ARGS__ }
#else

#define OPTKIT_PERFORMANCE_EVENTS(block_name, event_name, variable_name, ...)
#define OPTKIT_PERFORMANCE_BLOCK_EVENTS(block_name, event_name, variable_name, ...)

#endif
#endif
```

Figure 4.1: PMU Macro Definition

important thing is that these macros must be removable. We control this by checking the deployment configuration macro namely *CONF__PMU__MACROS__ENABLED*. Each measurement (RAPL, PMU etc.) has its own macro definitions for production.

```
#include <iostream>
#include <optkit.hh> // optkit header


int main()
{
    // ****** Init OPTKIT ********* //
    OPTKIT_RUNTIME;
```

```
    // ****** Embedded Freq Governor ********* //
    OPTKIT_FREQ_GOVERNOR;
    OPTKIT_FREQ_GOVERNOR(true); // -> data collector mode

    // ****** RAPL Monitoring ********* //
    OPTKIT_RAPL(variable_name, block_name);
    OPTKIT_RAPL_REPEAT(variable_name, block_name, count);

    // ****** PMU Event Monitoring ********* //
    OPTKIT_PERFORMANCE_EVENTS(block_name, event_name, variable_name, ...);
    OPTKIT_PERFORMANCE_BLOCK_EVENTS(block_name, event_name, variable_name, ...);

    // ****** Frequency Setting ********* //
    OPTKIT_SET_CPU_CORE_FREQ(frequency, socket);
    OPTKIT_SET_CPU_UNCORE_FREQ(frequency, socket);
    OPTKIT_SET_CPU_FREQ(core_freq,uncore_freq, socket);
    OPTKIT_RESET_CPU_FREQ(socket);
    OPTKIT_RESET_CPU_CORE_FREQ(socket);
    OPTKIT_RESET_CPU_UNCORE_FREQ(socket);

    // ****** Some High Level Performance Measurements ********* //
    OPTKIT_COMPUTE_AND_MEMORY_INTENSITY(variable_name,block_name);
    OPTKIT_COMPUTATIONAL_INTENSITY(variable_name, block_name);
    OPTKIT_CACHE_INTENSITY(variable_name, block_name);
    OPTKIT_DRAM_INTENSITY(variable_name, block_name);
    OPTKIT_IO_INTENSITY(variable_name, block_name);

    // ****** General Queries ********* //
    Query::<anything>;

    // ****** Frequency Related Queries ********* //
    QueryFreq::<anything>;

    // ****** PMU Related Queries ********* //
    QueryPMU::<anything>;
}
```

Listing 4.1: OPTKIT Building Blocks

```
#include <iostream>
#include <optkit.hh>

void foo() {
   OPTKIT_PERFORMANCE_EVENTS(block_name, pmu_events...);
   // code of interest 1
}
void boo() {
   OPTKIT_RAPL(block_name);
   // code of interest 2
}
void hoo(){
   OPTKIT_SET_CPU_FREQ(core_freq,uncore_freq,socket);
   {
       OPTKIT_RAPL(block_name); // code of interest 3
       {
         OPTKIT_PERFORMANCE_EVENTS(block_name2, pmu_events...);
         // code of interest 4
       }
       {
         OPTKIT_PERFORMANCE_EVENTS(block_name3, pmu_events...);
         // code of interest 5
       }
   }
   OPTKIT_RESET_CPU_FREQ(socket);
}

void main()
{
    OPTKIT_RUNTIME;
    // other code.
}
```

Listing 4.2: OPTKIT Example

It is important to note that every building block utilizes RAII. This means that upon its definition, it measures the block it belongs to. In addition, any of these building blocks can be used intrinsically. However, it should be kept in mind that they may slightly (can be disregarded) perturb the measurements when used in this way. An example of profiling using OPTKIT is provided in the listing[4.2].

In the foo() method, our focus lies on the performance of the code. Similarly, in the boo() method, our concern is the energy consumption of the relevant code.

It is also feasible to merge these concerns concurrently. In the hoo() method, we change the CPU frequency for our benefit while measuring the overall energy consumption of the entire method and maintaining our interest in certain events at a broader scope, and we progressively delve into specific events part by part.

Upon definition, these methods execute essential configurations and commence the measurement process. As they exit the scope at the end of the block, they cease measurements and export data in JSON format according to the specified configuration or just print them to the console by choice.

It should be noted that stacking these measurements on top of each other may diverge the actual results. In hoo() method, configuring performance counters and printing them out (as a JSON file or to console) may slightly disturb the energy consumption compared to a version in which these PMUs are not configured. Likewise, in-lining PMU counters may cause PMC multiplexing (due to lack of performance counters, these events are swapped in-out) resulting in slightly diverged results. Here the best practice would be to configure PMUs first thenRAPLand use *OPTKIT_PERFORMANCE_BLOCK_EVENTS* for grouping all PMU events and measuring them together without any multiplexing. Upon multiplexing, our tool prints a warning message to the user indicating that the results may be slightly diverged.

4.2 Installation and Using

OPTKIT employs Makefile as its build system and includes an auto-installation feature. After downloading from the GitHub page [34], the user only needs to

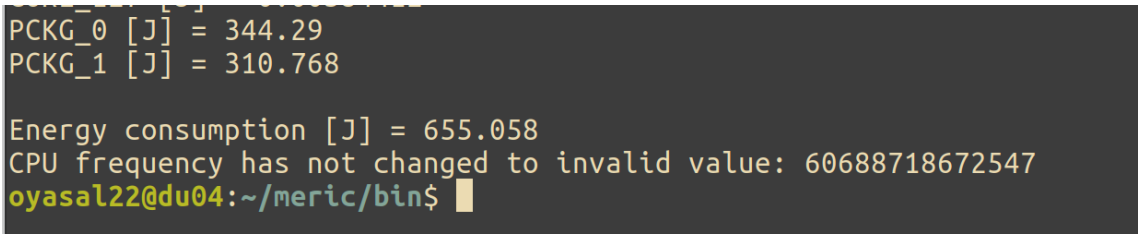


Figure 4.2: A100 RAPL Sleep MERIC Result

run *make install*. This command compiles all dependencies and the optkit tools, generates both static and dynamic library versions, and installs everything to the /usr/local/bin,lib,include directories. Once installed, the user can freely use OPTKIT in their projects. A simple example is given in 4.3. Considering the file name is "main.cc" the compilation command would be like

g++ main.cc -I/usr/local/include/optkit/ -I/usr/local/include/spdlog -fopenmp -loptkit -lspdlog -lpfm -ldl -lpthread -ltensorflow

4.3 Measurement Verification

In this part, we will display the accuracy of our tool in terms of energy consumption compared to the MERIC tool. The results for MERIC are given in Figure 4.2 and the results for OPTKIT is given in Figure 4.3 for the system's energy consumption during 4 seconds. While MERIC's mericStatic tool accumulates the results for a given program (sleep 4 in this case). OPTKIT's beacon_rapl prints out energy consumption each seconds for both sockects. When accumulated, we get 337.90 for socket 0 and 308.57 for socket 1 which is pretty close what MERIC reported.

4.4 Implementation Details

Optimizer Toolkit is developed using the vanilla C++11 standards. Although it supports some static methods for start-stop measurements and monitoring like *PAPI, MERIC* or any other profiling tool, the RAII programming idiom is heavily employed as the design choice for any kind of monitoring and measurements.

```
Package 0
        energy-pkg: 84.5494 Joules Consumed.
Package 1
        energy-pkg: 77.2026 Joules Consumed.

Package 0
        energy-pkg: 84.4568 Joules Consumed.
Package 1
        energy-pkg: 77.1202 Joules Consumed.

Package 0
        energy-pkg: 84.47 Joules Consumed.
Package 1
        energy-pkg: 77.1209 Joules Consumed.

Package 0
        energy-pkg: 84.4542 Joules Consumed.
Package 1
        energy-pkg: 77.135 Joules Consumed.
```

Figure 4.3: A100 RAPL Sleep OPTKIT Result

The functionality of OPTKIT is delivered through four primary classes: BlockProfiler, BlockGroupProfiler, RaplProfiler, and CPUFrequency, each offering fully customizable profiling configurations. To facilitate integration into development stages and to simplify configuration, we've wrapped these functionalities with C-style macros and provided default settings. This approach allows users to employ one of the core components with minimal knowledge or easily disable them as needed. The common building blocks of OPTKIT are summarized in Table [4.1]. It is worth noting that some of these building blocks like *probe, beacon*, governor* used OPTKIT as a library.

By default, when activated, RAPL will measure across all system sockets, PERFORMANCE_EVENTS will monitor the threads or cores utilized by the program, and CPUFrequency methods will operate on the specified socket numbers or core numbers provided as parameters. Upon exiting the corresponding block, they generate a JSON file named after the specified block_name and stores the results in that file. This behavior applies to both RAPL and PERFORMANCE profilers.

An example output is provided for the program given below. This program randomly init a large vector and accesses its element to find a sum.

```
#include <omp.h>
#include <optkit.hh>

#define VECTOR_SIZE 100000000 // 1 billion elements
#define NUM_ACCESSES 100000000 // 100 million random accesses
void random_access()
{
    OPTKIT_COMPUTE_AND_MEMORY_INTENSITY(ci_random_access, "random_access");
    // Initialize the vector with random values
    std::vector<double> vec(VECTOR_SIZE);

    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dis(0.0, 10.0); // Random values between
        0.0 and 10.0

    for (size_t i = 0; i <VECTOR_SIZE; ++i)
        vec[i] = dis(gen);
    double sum = 0.0;

#pragma omp parallel reduction(+ : sum)
    {
        std::mt19937 thread_gen(rd() ^ omp_get_thread_num()); // Seed
            differently for each thread
        std::uniform_int_distribution<> thread_dis(0, vec.size() - 1);

#pragma omp for
        for (int i = 0; i <NUM_ACCESSES; ++i)
        {
            int idx = thread_dis(thread_gen); // Get a truly random index
            sum += vec[idx];
        }
    }
    std::cout <<"Sum: " <<std::fixed <<sum <<std::endl;
```

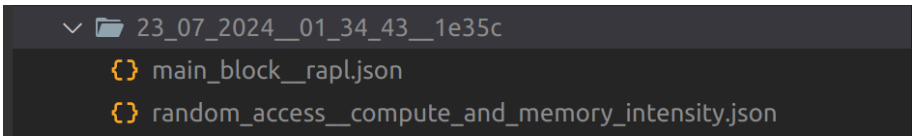


Figure 4.4: Example Case Program Folders

```
}


int32_t main(int32_t argc, char **argv)
{
    OptimizerKit optkit{};
    OPTKIT_RAPL(rapl_var, "main_block");
    random_access();
    return 0;
}
```

Listing 4.3: Example Case Program

When executed, OPTKIT generates a folder containing measurement JSON files. with the template 'dd_mm_yyyy__hh_mm_ss__guid(5)', For this program, the output is given in Figure 4.4. Since we monitor the intensity of the *random_access()* method and the energy consumption of whole program, it also generates 2 .json files. containing information for RAPL 4.7 and intensity 4.6. It should be noted that, tool exports the minimum amount of data and expect user to use OPTKIT's enrich tool to enrich this data. This is a design choice that aims OPTKIT to have a minimum overhead. When RAPL enriched with *optkit_enrich main_block__rapl.json* command, it extends the data with Figure 4.5. Same goes with the compute_and_memory_intensity, it generates data according to Figure 5.16b.

```
[
    {
        "readings": [
            {
                "duration": 2303.074951171875,
                "event_name": "rapl",
                "metrics_set": [
                    {
                        "description": "Consumed",
                        "metric_name": "edp",
                        "units": "Product",
                        "value": 97.10010832495988
                    },
                    {
                        "description": "Consumed",
                        "metric_name": "energy-total",
                        "units": "Joules",
                        "value": 42.16107177734375
                    },
                    {
                        "description": "Consumed",
                        "metric_name": "power",
                        "units": "Watt",
                        "value": 18.30642626541134
                    },
```


Figure 4.5: Example Case Enriched RAPL

```
[
    {
        "readings": [
            {
                "duration": 2273.009033203125,
                "event_name": "compute_and_memory_intensity",
                "metrics_set": [
                    {
                        "description": "Counted",
                        "metric_name": "instructions",
                        "units": "piece",
                        "value": 18504821250
                    },
                    {
                        "description": "Counted",
                        "metric_name": "flops_executed",
                        "units": "piece",
                        "value": 700000023
                    },
                    {
                        "description": "Counted",
                        "metric_name": "l1_l2_hits&misses__l3hit",
                        "units": "piece",
                        "value": 3318586989
                    },
                    {
                        "description": "Counted",
                        "metric_name": "l3_miss",
                        "units": "piece",
                        "value": 97187040
                    }
                ],
                "package_number": -1
            }
        ]
    }
]
```


Figure 4.6: Example Case Intensity Output

```
[
    {
        "readings": [
            {
                "duration": 2303.074951171875,
                "event_name": "rapl",
                "metrics_set": [
                    {
                        "description": "Consumed",
                        "metric_name": "energy-cores",
                        "units": "Joules",
                        "value": 34.24237060546875
                    },
                    {
                        "description": "Consumed",
                        "metric_name": "energy-gpu",
                        "units": "Joules",
                        "value": 0.810546875
                    },
                    {
                        "description": "Consumed",
                        "metric_name": "energy-pkg",
                        "units": "Joules",
                        "value": 41.956298828125
                    },
                    {
                        "description": "Consumed",
                        "metric_name": "energy-psys",
                        "units": "Joules",
                        "value": 0.20477294921875
                    }
                ],
                "package_number": 0
            }
        ]
    }
]
```


Figure 4.7: Example Case RAPL Output

Chapter 5

TOOL DEMONSTRATION

To demonstrate the capabilities of OPTKIT, we first explore *energy-aware strong scaling* to allocate the optimal amount of resources to maximize the energy efficiency of a given application, subsequently investigating co-scheduling options. We then investigate *dynamic frequency scaling* to identify the most effective core-uncore frequency pairs for optimal energy efficiency. Finally, we automate this process by implementing our custom embedded frequency governor, which utilizes a compact yet effective deep learning model and a callback mechanism to select the best frequency pair in run time, eliminating the need for prior application execution.

For our energy evaluations we used the term EDP (energy delay product) which is calculated as *Consumed Energy (Joules) x Time(seconds)*. Since the numbers may be large, we converted the energy in joules to kilojoules and put our results as K-EDP in our tables. In all charts, a lower K-EDP means better option in terms of energy efficiency and execution time compared to its counterparts.

As applications, we use *Rodinia*[4] *Polybench*[2] and *NAS*[1] benchmarks for our use cases. For our evaluations, we utilized 2-socket systems. As described in Section *3.2 (RAPL)*, we dedicated one socket exclusively to running our benchmarks, ensuring more precise measurements, while the other socket handled other tasks. In all cases, socket-1 is used to execute benchmarks. For all benchmarks, we utilise OPTKIT and its tool-set, while OPTKIT tools are used for visualisation, for bar charts, rapid-tables [3] is used. The specifications of the machine is given in the Table 5.1.

Machine/Spec	INTEL
OS	CentOS 7
KERNEL	3.10.0-1062.12.1.el7.x86_64
CPU	2 x Intel® Xeon Gold 6140 @ 2.3GHz
TOTAL CORES	72 (2 socket x 18 cores x 2 SMT)
ARCH	Skylake
MEM	Micron DDR4 4x16GB @ 2.6GHz
TDP	2 x 140W

Table 5.1: INTEL Machine Specifications

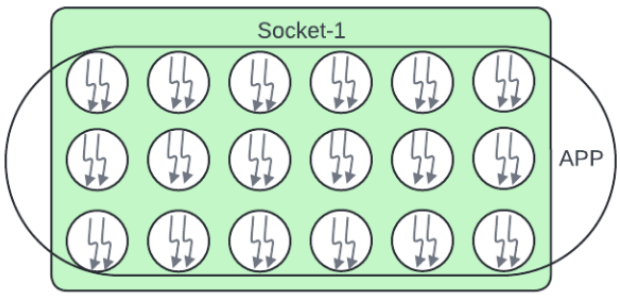


Figure 5.1: Strong Core Scaling Methodology

5.0.1 Energy-Aware Strong Scaling and Co-Scheduling

The starting point for this experiment is a straightforward observation that numerous parallel applications cannot be linearly parallelised. It is anticipated that as the number of cores increases in those applications, performance will reach to a saturation point after a certain number of cores while energy consumption continues to increase.

Our initial goal is to identify the ideal number of cores for a specific program and run programs using their optimal core count, while leaving other cores idle, thus resulting energy savings. Next step is to investigate whether co-scheduling applications can improve energy consumption.

Although it is expected that idle cores to consume less energy than active ones,

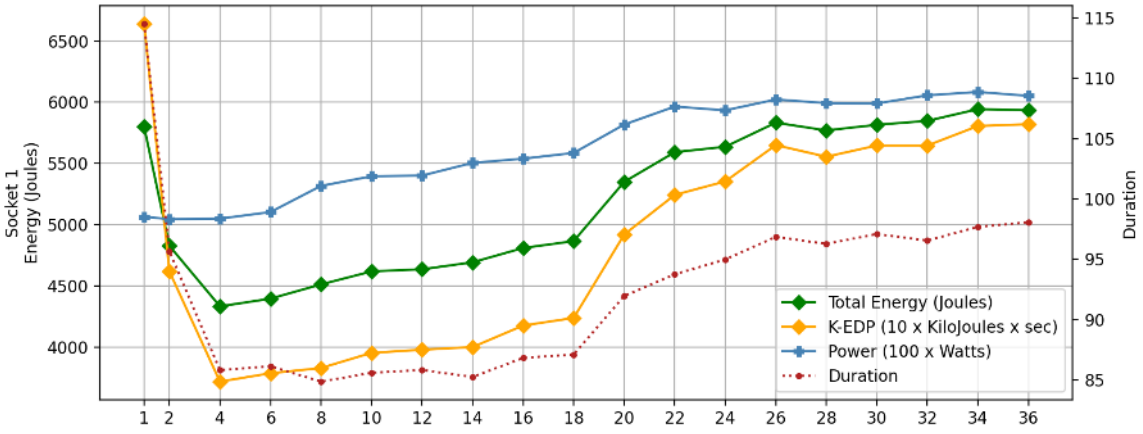


Figure 5.2: Hotspot3D Strong Scaling

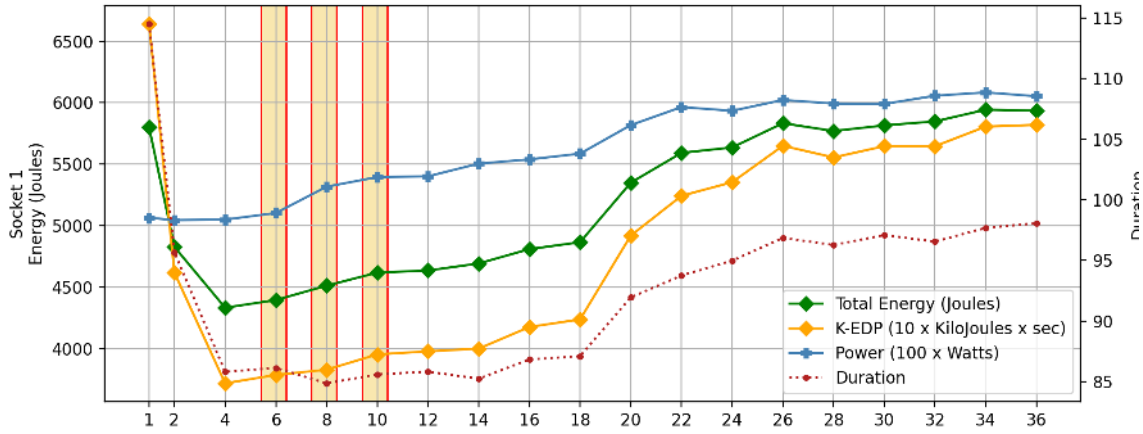


Figure 5.3: Hotspot3D Suggestion for Co Scheduling

our energy savings haven't been as promising as initially anticipated. In some cases, using 4 cores compared to 8-14 cores yields almost no benefit in terms of energy saving and performance. However, by choosing the optimum number of cores and then co-scheduling the application with another application, we can achieve energy savings.

Our setting for the core-scaling experiment is given in figure [5.1]. For each benchmark, we use OPTKIT's 'core_scaling' utility, which incrementally increases the core-thread count by one.

The strong scaling results of the Hotspot3D [5.2], Gemm [5.4], Heartwall [5.6] and LavaMD [5.8]benchmarks has been provided.

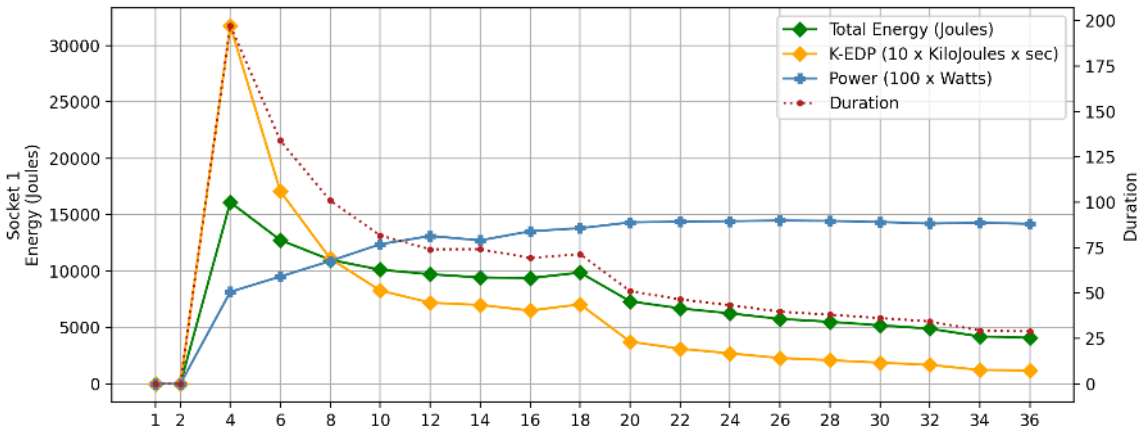


Figure 5.4: Gemm Strong Scaling
(cores 1&2 clipped out for readability)

Hotspot3D [5.2] reaches its peak performance with the lowest energy consumption at 8 cores. Interestingly, there is no significant performance difference between using 8 cores and 14 cores, although energy consumption slightly increases when using 14 cores. Considering intel provides the same frequency to each core regardless of whether they are used or not, this can be attributed to multi-threading overhead. Additionally, enabling SMT (increasing from 18 to 36 cores) significantly degrades both performance and energy efficiency for this benchmark.

Gemm in Figure 5.4 scales well with an increasing number of cores, with a notable performance boost occurring when SMT cores are utilized.

The heartwall in Figure 5.6 and LavaMD in Figure 5.8 scale well with an increasing number of cores; however, both do not gain as much from SMT.

optkit_vis_line has a parameter named *"–suggest"* that make a suggestion for co_scheduling based on a formula named co scheduling efficiency(CoSE)

$$\text{CoSE} = \frac{\text{K-EDP}}{\epsilon^4}.$$

Let ϵ be the core number such that

$$\epsilon = \begin{cases} \text{total\_core\_num - core\_num} & \text{if core\_num} < \frac{\text{total\_core\_num}}{2}, \\ \text{core\_num} & \text{otherwise.} \end{cases}$$

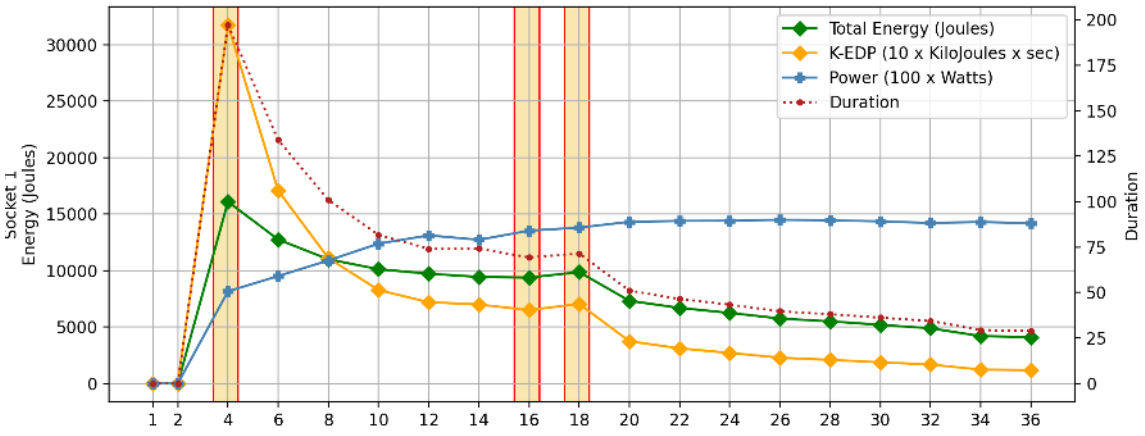


Figure 5.5: Gemm Suggestion for Co Scheduling
(cores 1&2 clipped out for readability)

In often cases lower core counts have higher K-EDP values. In a formula that we divide by core number without any modification, lower core counts take the highest values. To alleviate this situation we use mirroring by subtracting the maximum core number (if it is lower than the median, max physical core num) thus ϵ value is same for 2-34, 4-32, 16-20 etc. Note that the algorithm does not suggest a core number if its higher than the actual physical core number. Suggestions try to accumulate at the center if possible so a socket can be divided into two applications as shown in Figure 5.11 thus if K-EDP value is same for core 10 and 14, the CoSE value will be higher in with core 14 (it takes 22 as ϵ, 10 takes 26), however, the intuition is not guaranteed as shown in 5.3 since this is not the only metric that we look for.

The algorithm finds each CoSE vaule per core, then the value at T_nCoSE is suggested if $T_nCoSE > T_{n-1}CoSE$. Users get 3 suggestions at max by default, which can be changed by passing another parameter. Each suggested core number is $<$ # of physical core count (in this case, up to the 18th core).

Suggestions for lavaMD, Gemm, Hotspot3D, Heartwall given in Figures 5.9 5.5 5.3 5.7 Note that these are just suggestions, not ground truths.

Lastly, all figures illustrate a strong correlation between energy consumption and time, confirming the validity of the "race-to-sleep" factor.

As a next step, we elaborate the initial setting in Figs. 5.1 to 5.10 and find out

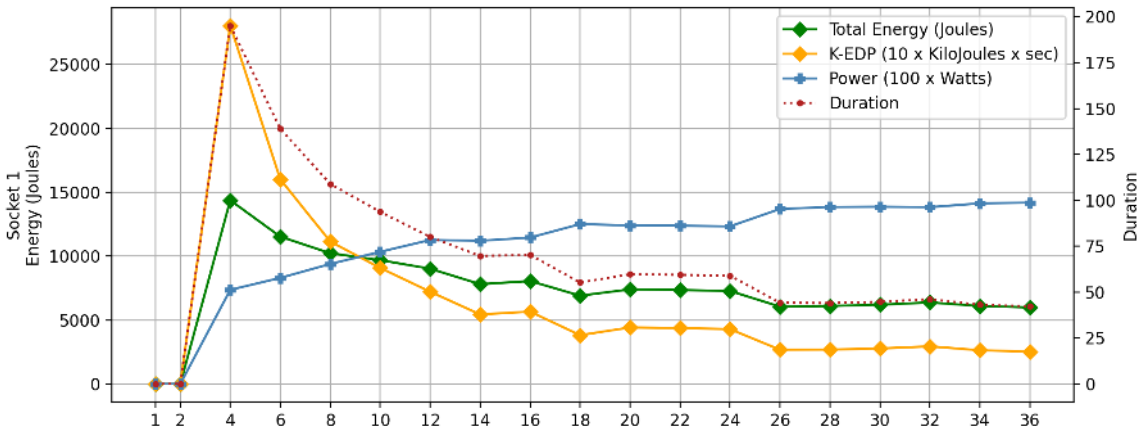


Figure 5.6: Heartwall Strong Scaling
(cores 1&2 clipped out for readability)

to what extent the SMT and the available resources affect the benchmarks. Then for all the combinations we concluded the total K-EDP for sequential execution of these benchmarks.

Based on the results from single executions [5.12] it is observed that dedicating all available resources to applications generally leads to better energy savings in 3 out of 4 benchmarks. The same trend is observed in sequential executions [5.13] Notably, while some applications benefit from Simultaneous Multithreading (SMT) (case 2), others show improved performance with one thread per core execution (case 3). It should be noted for sequential executions k-edp value is calculated as *total energy x total time*.

$$K\text{-EDP} = \left(\sum_{\mathrm{E}} Bench1, Bench2\right) \cdot \left(\sum_{\mathrm{T}} Bench1, Bench2\right)$$

The methodology for co-scheduled execution is given in Figure 5.11. The K-EDP values shown in this graph represent the total execution. In other words, we begin by running two benchmarks simultaneously on the same socket according to our methodology. After both applications have completed their execution, we then obtain the K-EDP value from the benchmark that last longer. When compared to serial execution, co-scheduled versions achieve a K-EDP benefit ranging from 44% to 60% in all cases.

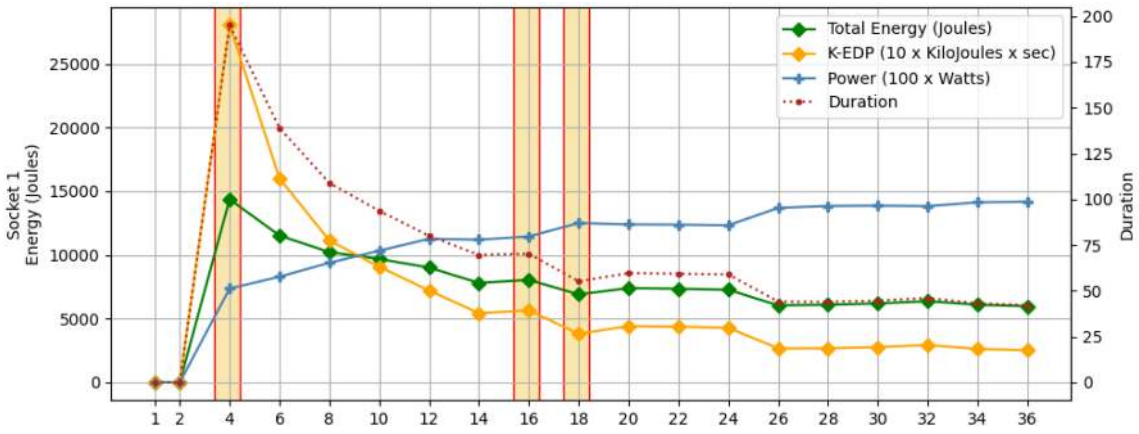


Figure 5.7: Heartwall Suggestion for Co Scheduling
(cores 1&2 clipped out for readability)

5.0.2 Dynamic Frequency Scaling Effects

In this section, we integrated core and uncore frequency change utilities into our tool as discussed in the section *CPU Frequency*. We then used OPTKIT's freq_scaling utility to scale the CPU core and uncore frequencies then used vis_freq_heatmap utility to create a heat map, illustrating the optimal frequencies in terms of EDP for each benchmark. This helped us determine the best core-uncore frequency combination for the overall energy efficiency of a given benchmark without changing any code.

Benchmark	Default K-EDP	Optimum K-EDP	Best Frequency (core - uncore)	K-EDP Savings	Default - Optimum Duration (Seconds)
Gemm	118.54	110.91	2.301 - 1.8	6.43%	28.90 — 28.04
Hotspot3D	582	332.42	2.301 - 1.2	42.88%	98.06 — 80.82
LavaMD	462.98	433.54	2.301 - 1.2	6.35%	57.29 — 55.60
Heartwall	252.36	220.42	2.301 - 1.2	12.65%	42.18 — 42.95

Table 5.2: Freq Scaling Optimised Result Comparison

An example of a heatmap for the gemm benchmark is given in Figure 5.15. As

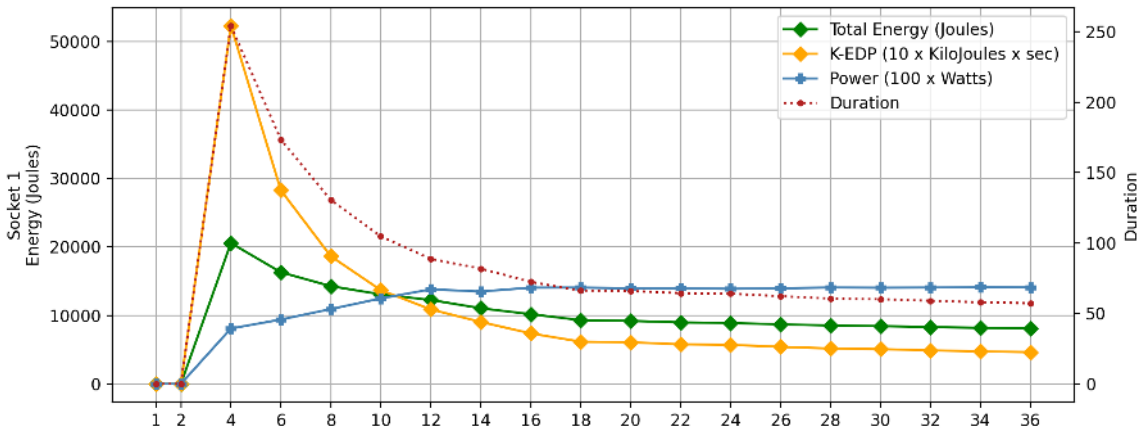


Figure 5.8: LavaMD Strong Scaling
(cores 1&2 clipped out for readability)

can be seen, the best frequency pair for gemm benchmark is 2.301 (core turbo) and 1.8 (uncore). We then repeated these steps for other benchmarks and obtained the results presented in Table [5.2]. Since the CPU frequency governor typically defaults to using maximum core and uncore frequencies, adjusting only the uncore frequency allowed us to achieve up to 42% savings without compromising performance. Although this approach is attractive, it is mostly impractical because it requires running benchmarks for every possible core-uncore frequency pair, and not all benchmarks complete quickly some event take a day to complete.

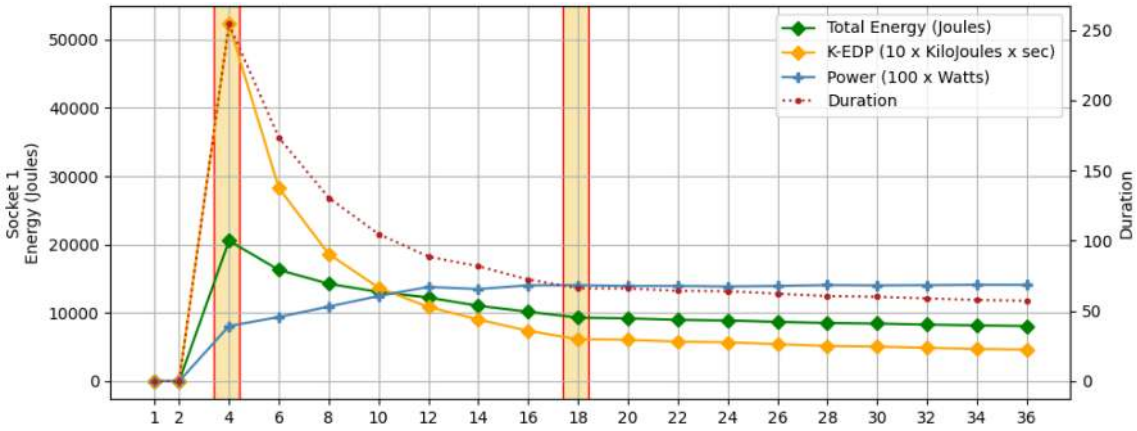


Figure 5.9: LavaMD Suggestion for Co Scheduling
(cores 1&2 clipped out for readability)

5.0.3 Custom Frequency Governor

In this section, we eliminate the need for the brute-force approach to find the best frequency pair to achieve the best energy savings by integrating a custom frequency governor directly into OPTKIT. As of this writing, there are six frequency governors in Linux systems: performance, powersave, userspace, ondemand, conservative, and schedutil. Details about these governors can be found in Table 3.2. Among these governors, ondemand and conservative are notable for dynamically adjusting the frequency based on the current CPU load. However, this load is task-agnostic, meaning that the CPU frequency can be increased during a heavy I/O operation, even though the CPU cores only play a role in initiating system calls and not in calculations or other tasks. As indirectly demonstrated in the previous section, current frequency governors are insufficient for optimal energy savings. Therefore, we believe that there is potential for improvement in this area.

To alleviate the drawbacks given in the previous section, we devise a better approach by finding the best frequency as the program runs. The mechanism of our method is given in the Figure 5.16a.

During run-time, the state of an application evolves over time. At a given time T_1, the computational intensity may be high, with numerous floating point operations occurring. By T_2, the computation results might be written back to a variable,

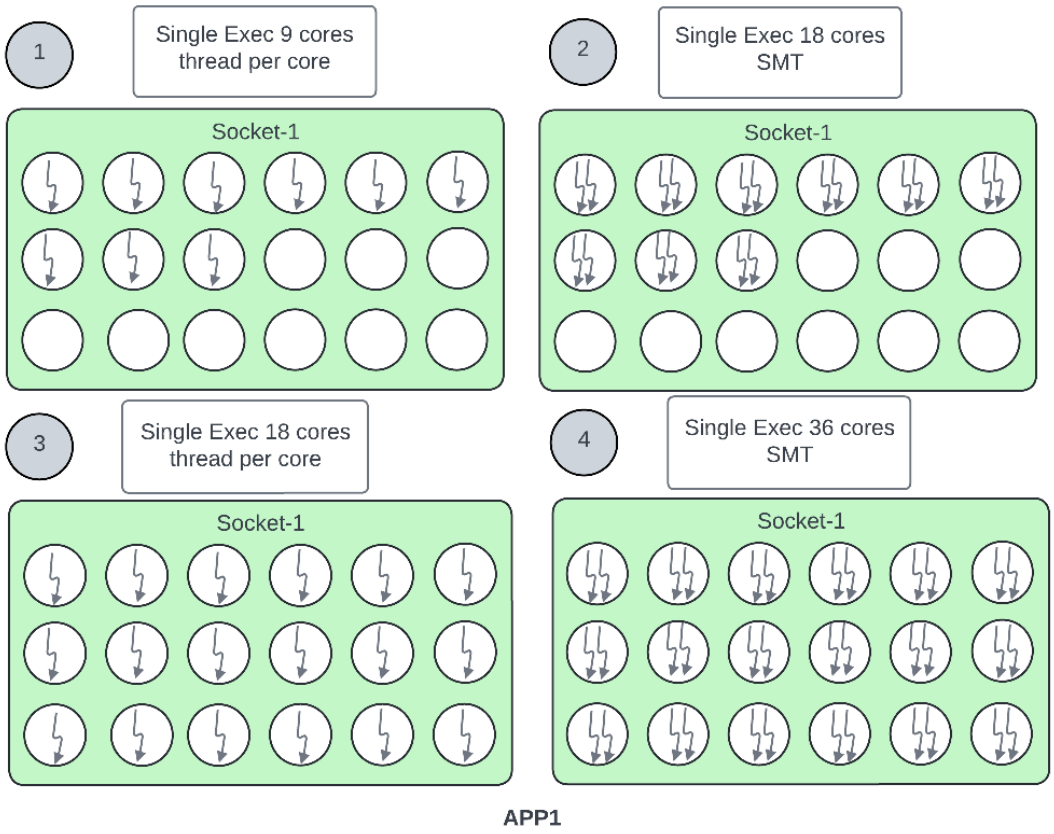


Figure 5.10: Strong Core Scaling Detailed Methodology

leading to higher cache and memory intensities. Additionally, there might be a time T_3 when a file is read and transmitted over a network connection or there may be a time T_4 all these happening at the same time.

Our approach involves first implementing the given mechanism [5.16a] that uses PMUs to monitor and record all these state changes. We then selected benchmarks from the Polybench [2] and Rodinia [4] suites and used brute-force approach to find their best frequency pairs and saved the application states to construct a training dataset for our deep learning model. Later, we use this model for other applications to determine the best frequency during runtime.

In the methodology[5.16a] , we set up performance counters to measure three different metrics[5.16b]: computational intensity, cache intensity, and memory intensity. Computational intensity is defined as the total number of floating-point

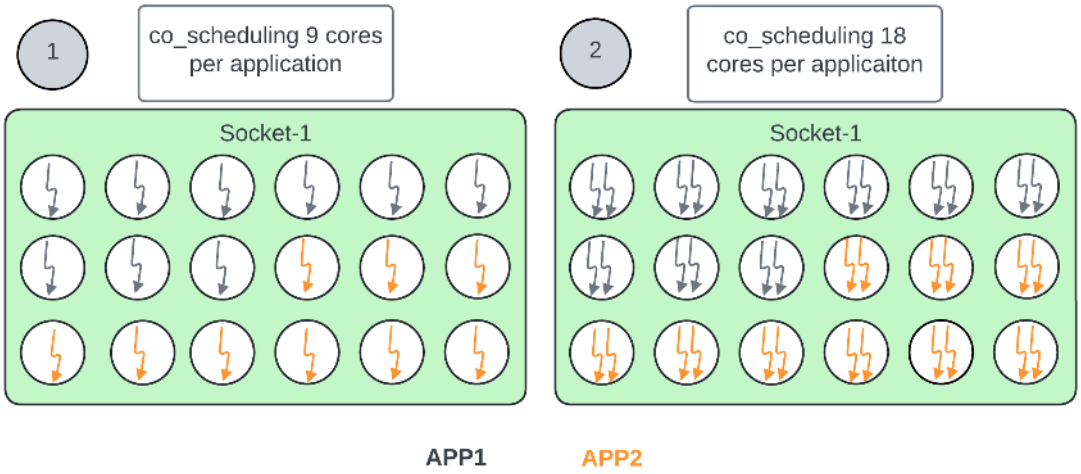


Figure 5.11: Co Scheduling Methodology

operations performed before a single DRAM request is made. The cache intensity is calculated as the total number of cache hits and misses divided by the total number of retired instructions, excluding LLC cache misses because they are counted as DRAM requests. Memory intensity is measured by the number of LLC cache misses relative to the number of retired instructions. We did not include IO intensity as input to our model; however, OPTKIT can detect any file operation and optionally perform an operation as a response. This feature is implemented via LD_PRELOAD feature.

The way we record the state of the benchmarks we selected and how we enable our custom governor in other benchmarks that we test is shown in listing[5.1]. Once enabled, performance counters create a callback to our governor approximately each 3 seconds (this period can be altered by the user), then our custom governor reads the 3 Intensities described above, then sets to our model to predict the new frequency pair. If the difference between the current frequency and the new suggested frequency is greater than our threshold frequency (0.2GHZ), then the frequency is set. else it is disregarded.

The frequency scaling results for the benchmarks used to create the dataset are shown in Figure 5.17.

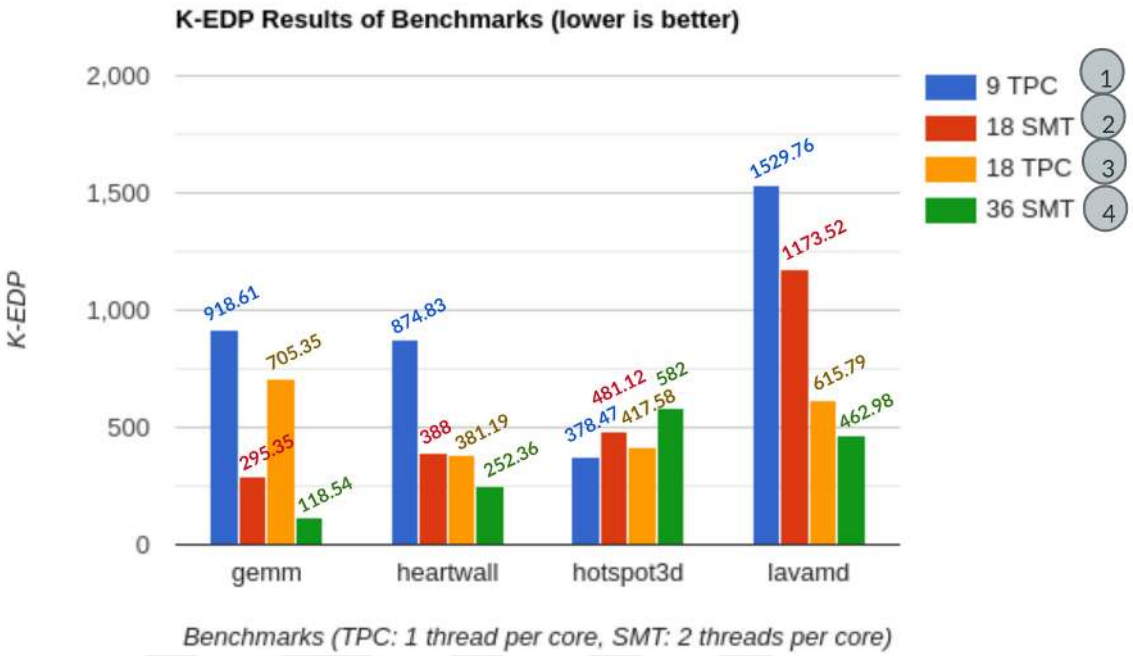


Figure 5.12: Single Execution Results for Detailed Methodology

```
#include <iostream>
#include <optkit.hh>

void main(int argc, char** argv)
{
    OPTKIT_RUNTIME;
    OPTKIT_RAPL(main_block,argv[0]);
    OPTKIT_FREQ_GOVERNOR;
    //OPTKIT_FREQ_GOVERNOR(true); // to record application state

    // rest of the code
}
```

Listing 5.1: OPTKIT Custom Governor Setting

The loss ratio of our DL model is 0.0034 while MSE = (0.003987, 0.005490) for core-uncore frequencies. When we enable OPTKIT's embedded governor on benchmarks Gemm, Hotspot3D, LavaMD and Heartwall, we see some improvements regarding K-EDP compared to normal execution. Then to see if our model can be

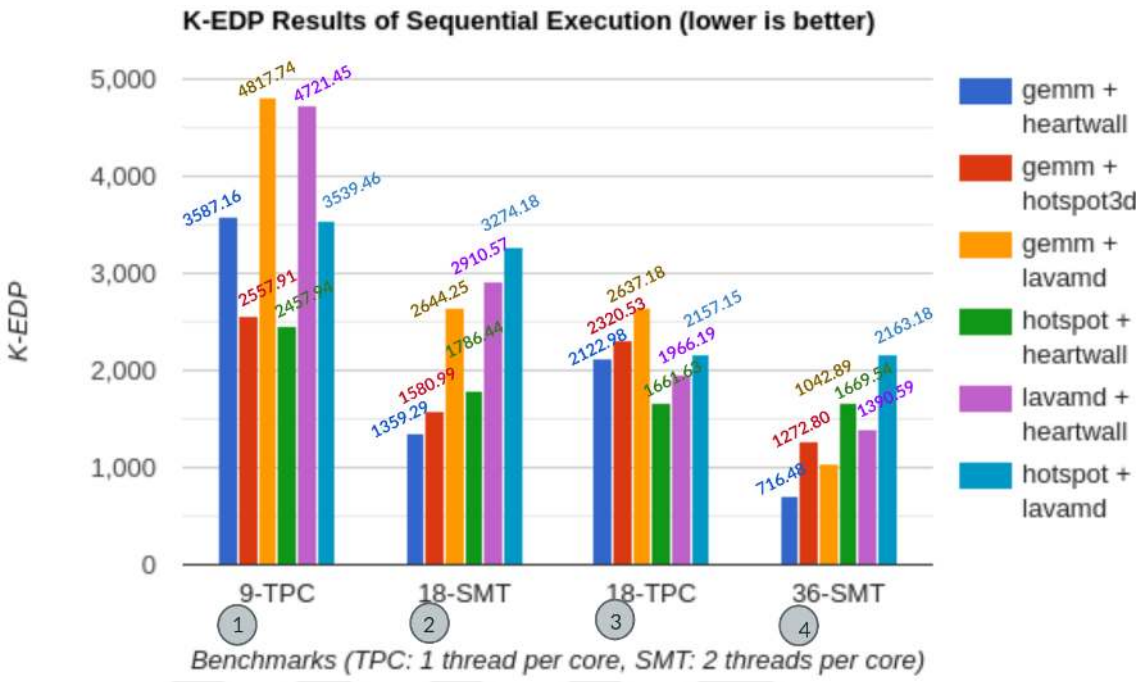


Figure 5.13: Sequential Execution Results for Detailed Methodology

used in other benchmarks, we tested on NAS [1] benchmarks. Results are given in the Table 5.3.

Based on the results, OPTKIT's runtime optimization improves K-EDP and energy savings for all benchmarks except one outlier (ep.C). Although the K-EDP savings for Gemm, Heartwall, and LU.C are not very significant, Heartwall and LU.C still achieve energy savings of 22.65% and 6.51%, respectively. Except some benchmarks, the overhead is at the system noise level. It is evident that we can achieve higher energy efficiency with little to no sacrifice in performance.

Overall, we can conclude that with just three lines of code, OPTKIT can enhance the energy efficiency of your application without any work in advance.

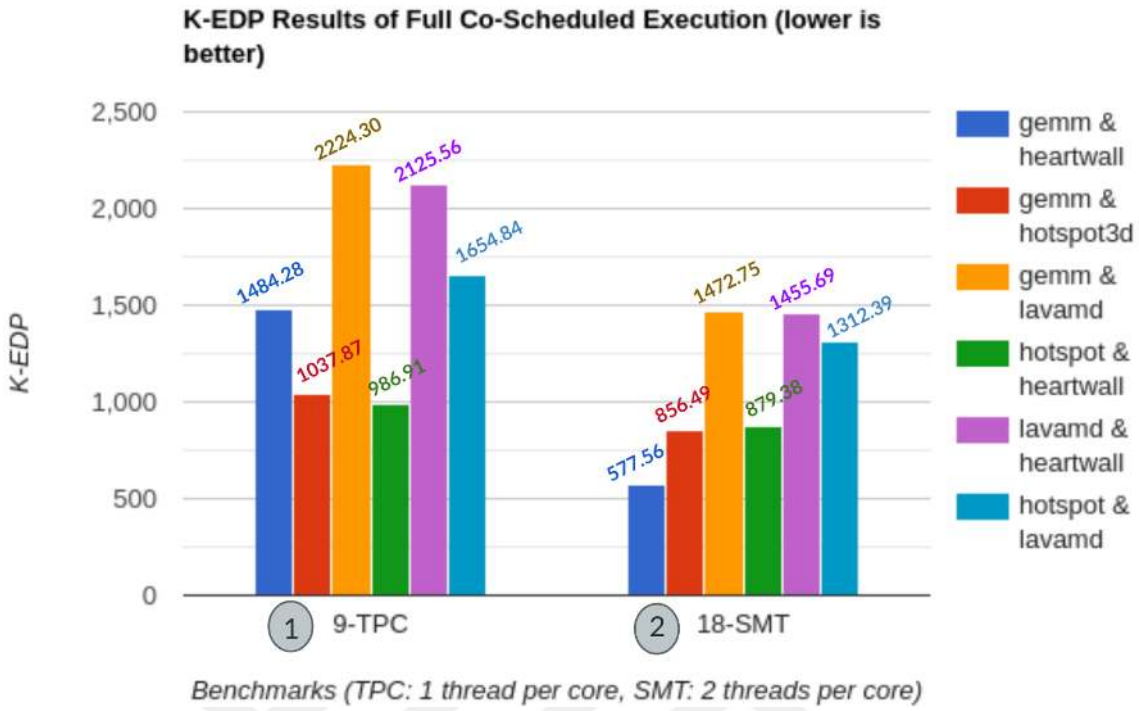


Figure 5.14: Co-Scheduled Execution Results for Detailed Methodology

5.0.4 Cross Platform Execution Results

In this section, we reproduced the results by using an AMD machine. Its specs can be found in the Table 5.4. Unfortunately, AMD sockets do not provide an interface to programmatically change the uncore frequency like intel does; otherwise, we could not find this feature. Given the reason, we disregarded the frequency use case for the AMD processor.

Cross Platform Energy-Aware Strong Scaling and Co-Scheduling

For all use-cases we follow the same settings (use all cores for scaling, use half of the cores with 1 thread and SMT threads for co-scheduling) given for Intel, but of course according to the processor.

Gemm core scaling results for new AMD machines can be seen in Fig 5.18. In A100 machine, we see performance improvements up to the 26th core. When SMT is enabled, there is a slight degradation in performance, but this effect levels off as the number of cores increases. We got similar results on our previous Intel experiment

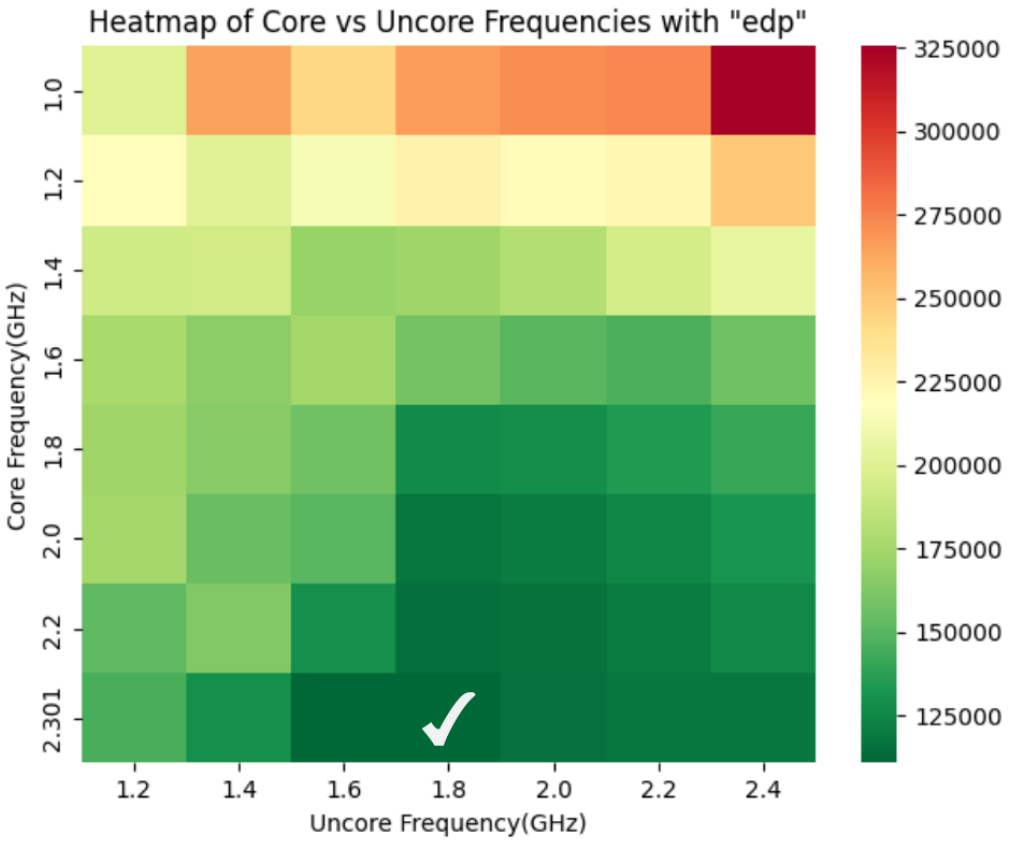


Figure 5.15: Heat-Map Gemm

with the difference that SMT cores had a good benefit in energy and performance.

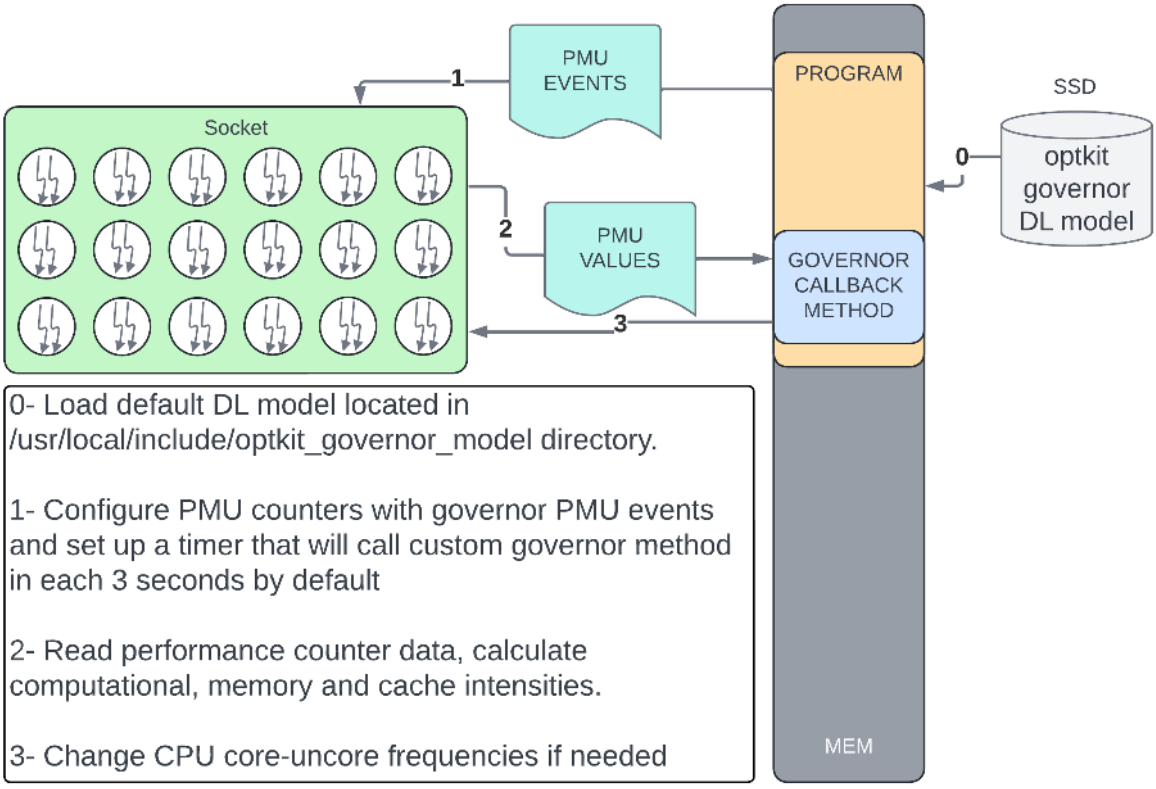
In heartwall core scaling in the Fig 5.20. In the A100 machine, the SMT cores do not affect performance or energy efficiency. However, we observe two significant drops in energy efficiency and performance at the 24th and 50th cores. After 52th cores, there is no change in energy and performance along with the power, indicating that the app reaches its saturation point. We also have a similar result with Intel.

The Hotspot3D core scaling is shown in Figure 5.21 and the Lava core scaling in Figure 5.19. With SMT cores, we get a slight decrease in the A100 machine similar to our initial Intel result.

The sequential execution of the parallel benchmarks is given in Table 5.5, while the co-scheduling results of the A100 machine are given in Table 5.6. The trends observed in both sequential and co-scheduled executions (whether TPC is optimized for SMT or not) are akin to those seen with Intel. Similar to Intel, where 4 out of

6 applications benefit from co-scheduling, 3 out of 6 applications on the A100 also show benefits.

The benefits of co-scheduling applications have been extensively covered in other papers and studies. However, we simply demonstrate that with our tool's co_schedule utility, you can effortlessly co-schedule applications according to your desired settings and compare the results to find the best match with minimal effort.



(a) Custom Governor Structure

$$Computational\ intensity = \frac{\text{Flops Executed}}{Total\ number\ of\ L3\ cache\ misses}$$

$$\textit{Cache Intensity} = \frac{\text{Total \# of L1| L2| L3 cache hits \& L1|L2 misses}}{\text{\# of instructions}}$$

$$\textit{Memory Intensity} = \frac{\text{Total \# of L3 cache \& misses}}{\text{\# of instructions}}$$

(b) Custom Governor Parameters

Figure 5.16: Overview of Custom Governor and its Parameters

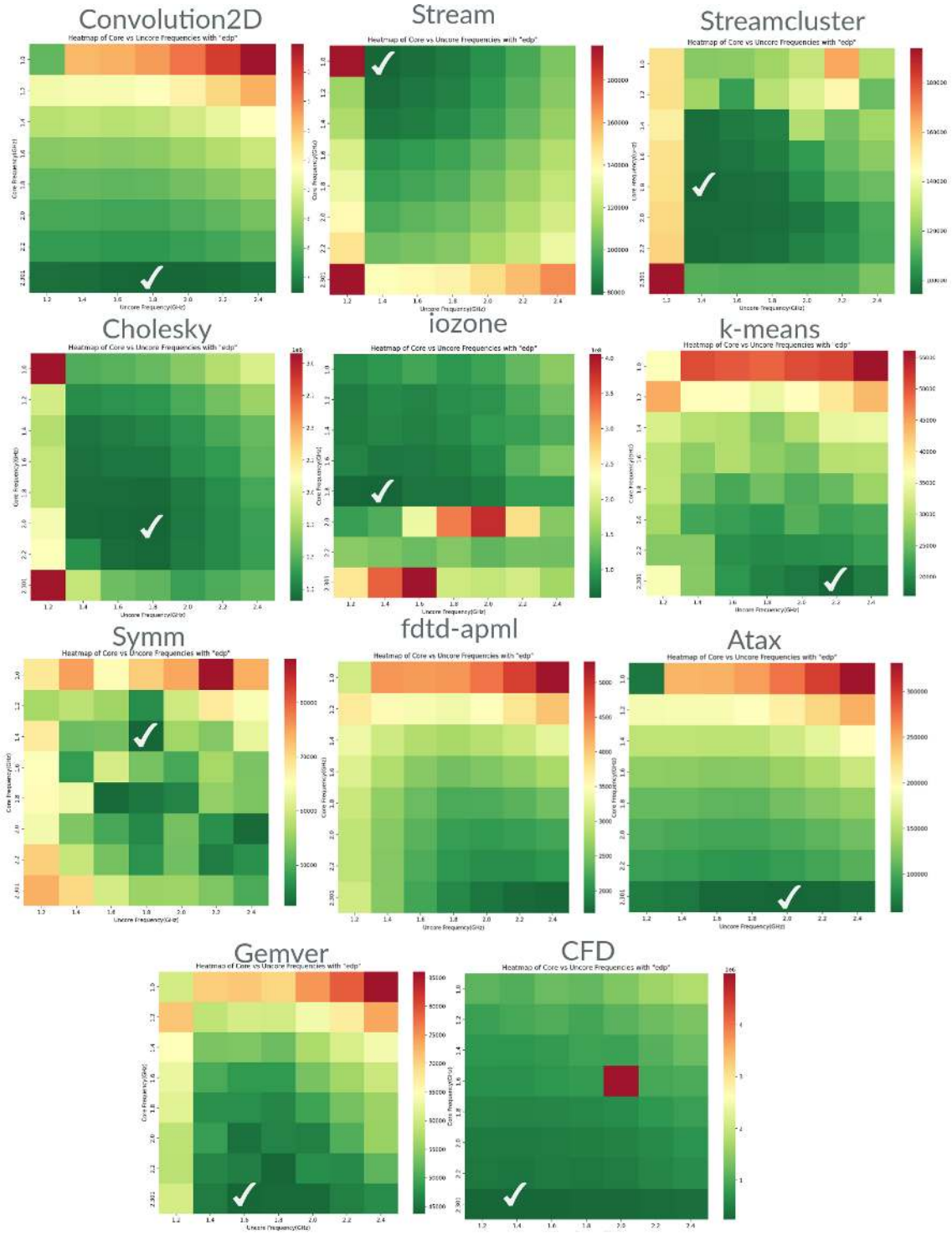


Figure 5.17: Frequency Scaling Results of Model Benchmarks

Benchmark	K-EDP	Energy (J)	Power(W)	Core(GHZ)	Uncore(GHZ)	K-EDP Savings	Energy Savings	Duration(seconds)
Gemm	118.54 —117.16	4101.09 — 4071.50	141.88— 141.49	2.301 —2.301	2.4 —2.1	1.16%	0.72%	28.90 — 28.92
Hotspot3D	582 — 458.18	5935.38 — 5633.04	60.52 — 69.25	2.301 —2.301	2.4 —2.4	21.41%	5.09%	98.06 —98.33
LavaMD	462.98 — 443.75	8081.47 —7887.60	141.06 —140.20	2.301 —2.301	2.4 —1.3	4.15%	2.39%	57.29 — 56.25
Heartwall	252.36 — 252.51	5982.85 —4627.19	141.83 — 84.79	2.301 —2.3	2.4 —1.3	-0.059%	22.65%	42.18 —54.57
bt.B	1029.55 — 894.53	11090.76 — 9692.21	119.47 — 105.01	2.301 — 2.301	2.4 — 1.6	13.11%	12.61%	92.55 —92.29
cg.C	76.97 — 55.95	3107.64 —2118.34	125.46 — 80.19	2.301 — 1.6	2.4 — 2.0	27.30%	31.83%	24.76 — 26.41
ep.C	32.59 —103.40	2098.92 — 2195.09	135.17 — 46.59	2.301 — 1.0	2.4 — 1.2	-217.27%	-4.58%	15.52 —47.10
lu.C	604.63 — 604.45	9240.84 —8639.18	141.23 —123.47	2.301 — 2.301	2.4 — 1.6	0.02%	6.51%	65.43 —69.96
sp.C	18568.45 —15621.02	46175.49 —33364.37	114.82 —71.26	2.301 — 2.0	2.4 — 1.9	15.87%	27.74%	402.12 —468.19

Table 5.3: Runtime Optimisation Results (normal exec — optkit runtime)

Machine/Spec	INTEL
OS	Ubuntu 20.04.6
KERNEL	5.15.0-71-generic
CPU	AMD EPYC 7742 (256) @ 2.250GHz
TOTAL CORES	256 (2 socket x 64 cores x 2 SMT)
ARCH	EPYC
MEM	Micron DDR4 4x16GB @ 2.6GHz
TDP	2 x 225W

Table 5.4: AMD Machines for Cross Platform Execution

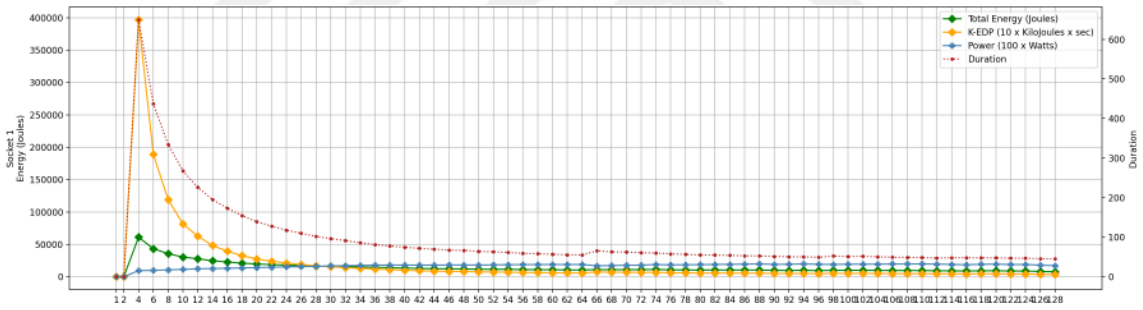


Figure 5.18: A100 Machine Gemm Core Scaling

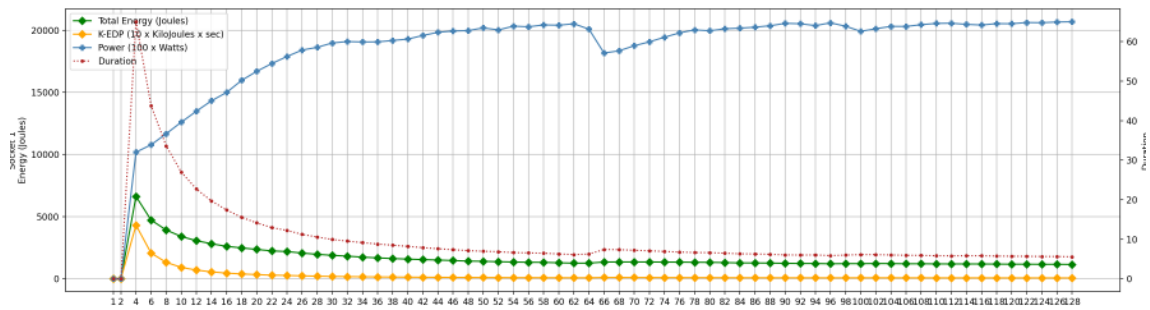


Figure 5.19: A100 Machine LavaMD Core Scaling

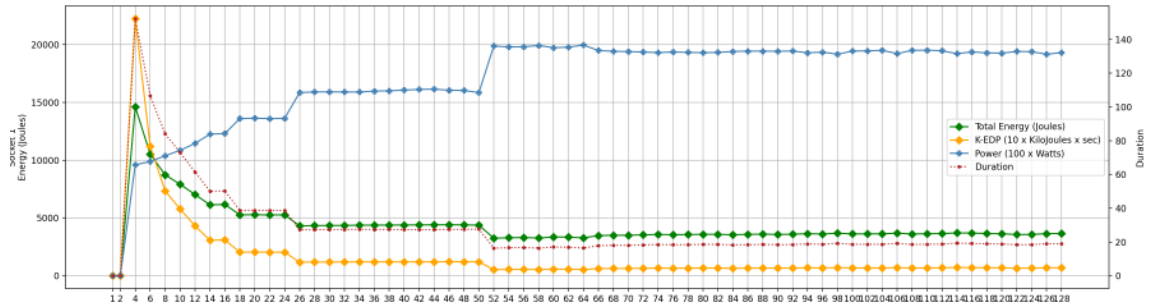


Figure 5.20: A100 Machine Heartwall Core Scaling

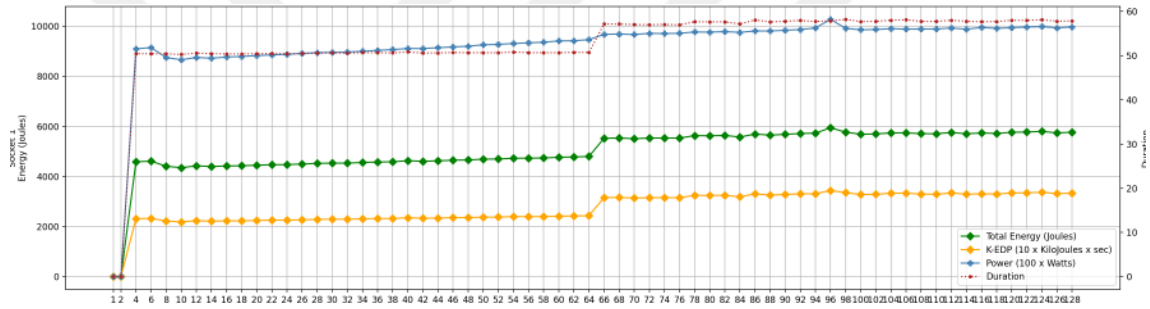


Figure 5.21: A100 Machine Hotspot3D Core Scaling

Benchmarks	**64-TPC**	**128-SMT**	**Savings %**
	Duration - K-EDP	**Duration - K-EDP**	
Gemm + Heartwall	71.65 - 981.50	64.04 - 728.85	10.62 — 25.74
Gemm + Hotspot3D	67.13 - 1612.11	102.95 - 1389.0	-53.35 — 13.83
Gemm + LavaMD	61.47 - 717.78	50.66 - 449.57	17.58 — 37.36
Hotspot3D + Heartwall	67.13 - 542	76.78 - 723.69	-14.37 — -33.52
Hotspot3D + LavaMD	56.96 - 344.94	63.40 - 438.63	-11.30 —-27.16
Heartwall + LavaMD	22.67 - 102.87	24.49 - 117.75	-8.02 — -14.46

Table 5.5: A100 Sequential Execution Results

Benchmarks	**32-TPC**	**64-SMT**	**Savings %**
	Duration - K-EDP	**Duration - K-EDP**	
Gemm & Heartwall	93.24 -1596.48	78.68 - 1082.49	15.61 — 32.19
Gemm & Hotspot3D	90.85 - 1444.87	75.72 - 943.07	16.65 — 34.72
Gemm & LavaMD	91.86 - 1478.67	75.42 - 978.27	17.89 — 33.83
Hotspot3D & Heartwall	52.03 - 368.04	59.15 - 444.53	-13.68 — -20.78
Hotspot3D & LavaMD	51.87 - 290.95	58.79 - 375.58	-13.34 — -29.08
Heartwall & LavaMD	29.27 - 154.43	23.28 - 106.30	20.46 — 31.16

Table 5.6: A100 Co-Scheduling Results

Chapter 6

CONCLUSION

In this work, we present OPTKIT, a dedicated tool/API designed for energy analysis and optimization of the runtime of CPU software. The use cases demonstrated in this research highlight the tool's functionality and user-friendliness. While the focus is primarily on energy efficiency, OPTKIT is also well-suited for performance analysis of code blocks. This study not only deepens our understanding of energy-efficient software development but also contributes to the advancement of tools that emphasize sustainability in the fast-evolving computing technology landscape. As the industry increasingly prioritizes energy-conscious computing, OPTKIT stands out as a valuable asset in promoting more efficient and sustainable software practices.

Chapter 7

FUTURE WORK

While we currently offer some abstracted performance metrics, we will be incorporating Top Down Micro-architecture Analysis (TMAM) into our code-base to provide users with a comprehensive performance analysis. Considering the current widespread use of AI and the Python language, we plan to broaden our tool to be compatible with Python3. Additionally, we will explore the potential to extend our embedded governor to other applications and possibly develop a new frequency governor for Linux systems that optimizes energy savings out of the box.

BIBLIOGRAPHY

[1] Nas benchmark.

[2] Polybench benchmark.

[3] Rapid tables.

[4] Rodinia benchmark.

[5] *Com Detective: A Lightweight Communication Detection Tool for Threads*. Aperta, January 2019.

[6] Mohammad Aldossary and Karim Djemame. Energy consumption-based pricing model for cloud computing. 09 2016.

[7] Mohammad Aldossary, Karim Djemame, Ibrahim Alzamil, Alexandros Kostopoulos, Antonis Dimakis, and Eleni Agiatzidou. Energy-aware cost prediction and pricing of virtual machines in cloud computing environments. *Future Generation Computer Systems*, 93, 11 2018.

[8] Lukas Alt, Anara Kozhokanova, Thomas Ilsche, Christian Terboven, and Matthias S. Mueller. An experimental setup to evaluate rapl energy counters for heterogeneous memory. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*, ICPE '24, page 71–82, New York, NY, USA, 2024. Association for Computing Machinery.

[9] https://www.amd.com/en/developer/uprof.html.

[10] https://www.vi-hps.org/tools/archer.html.

[11] https://azure.microsoft.com/en-us/pricing/purchase-options/pay-as-you-go/.

[12] https://valgrind.org/docs/manual/cl-manual.html.

[13] Anamika Chowdhury, Madhura Kumaraswamy, and Michael Gerndt. Readex tool suite for energy-efficiency tuning of hpc applications. In *Proceedings of the 2017 Workshop on Software Engineering Methods for Parallel and High Performance Applications*, SEM4HPC '17, page 11–12, New York, NY, USA, 2017. Association for Computing Machinery.

[14] Stefano Corda, Bram Veenboer, and Emma Tolley. Pmt: Power measurement toolkit. *arXiv*, 2022:1–3, November 2022.

[15] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. Rapl: memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '10, page 189–194, New York, NY, USA, 2010. Association for Computing Machinery.

[16] Armen Dzhagaryan and Aleksandar Milenković. Impact of thread and frequency scaling on performance and energy in modern multicores: a measurement-based study. In *Proceedings of the 2014 ACM Southeast Regional Conference*, ACM SE '14, New York, NY, USA, 2014. Association for Computing Machinery.

[17] Neha Gholkar, Frank Mueller, and Barry Rountree. Power tuning hpc jobs on power-constrained systems. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, page 179–191, New York, NY, USA, 2016. Association for Computing Machinery.

[18] David Guyon. *Supporting energy-awareness for cloud users.* PhD thesis, 12 2018.

[19] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring energy consumption for short code paths using rapl. *SIGMETRICS Perform. Eval. Rev.*, 40(3):13–17, jan 2012.

[20] Can Hankendi and Ayse K. Coskun. Reducing the energy cost of computing through efficient co-scheduling of parallel workloads. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, page 994–999, San Jose, CA, USA, 2012. EDA Consortium.

[21] Ranjan Hebbar and Aleksandar Milenković. Pmu-events-driven dvfs techniques for improving energy efficiency of modern processors. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 7(1), aug 2022.

[22] Song Huang, Michael Lang, Scott Pakin, and Song Fu. Measurement and characterization of haswell power and energy consumption. In *Proceedings of the 3rd International Workshop on Energy Efficient Supercomputing*, E2SC '15, New York, NY, USA, 2015. Association for Computing Machinery.

[23] https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2024-1/overview.html.

[24] Abhishek Jaiantilal, Yifei Jiang, and Shivakant Mishra. Modeling cpu energy consumption for energy efficient scheduling. In *Proceedings of the 1st Workshop on Green Computing*, GCM '10, page 10–15, New York, NY, USA, 2010. Association for Computing Machinery.

[25] Jaimie Kelley, Christopher Stewart, Devesh Tiwari, and Saurabh Gupta. Adaptive power profiling for many-core hpc architectures. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*, pages 179–188, 2016.

[26] Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K. Nurminen, and Zhonghong Ou. Rapl in action: Experiences in using rapl for power measurements. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(2), mar 2018.

[27] Kashif Nizam Khan, Filip Nybäck, Zhonghong Ou, Jukka K. Nurminen, Tapio Niemi, Giulio Eulisse, Peter Elmer, and David Abdurachmanov. Energy profiling using igprof. In *Proceedings of the 15th IEEE/ACM International Sympo-*

sium on Cluster, Cloud, and Grid Computing, CCGRID '15, page 1115–1118. IEEE Press, 2015.

[28] Shin-gyu Kim, Chanho Choi, Hyeonsang Eom, Heon Y. Yeom, and Huichung Byun. Energy-centric dvfs controling method for multi-core platforms. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 685–690, 2012.

[29] Simone Libutti, Giuseppe Massari, Patrick Bellasi, and William Fornaciari. Exploiting performance counters for energy efficient co-scheduling of mixed workloads on multi-core platforms. In *Proceedings of Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms*, PARMA-DITAM '14, page 27–32, New York, NY, USA, 2014. Association for Computing Machinery.

[30] Peini Liu, Gusseppe Rocca, and Jordi Guitart. *Energy-Aware Dynamic Pricing Model for Cloud Environments*, pages 71–80. 11 2019.

[31] Charalampos Marantos, Lazaros Papadopoulos, Christos P. Lamprakos, Konstantinos Salapas, and Dimitrios Soudris. Bringing energy efficiency closer to application developers: An extensible software analysis framework. *IEEE Transactions on Sustainable Computing*, 8(2):180–193, 2023.

[32] https://github.com/LLNL/msr-safe.

[33] Anilkumar Nandamuri, Abid M. Malik, Ahmad Qawasmeh, and Barbara M. Chapman. Power and energy footprint of openmp programs using openmp runtime api. In *Proceedings of the 2nd International Workshop on Energy Efficient Supercomputing*, E2SC '14, page 79–88. IEEE Press, 2014.

[34] https://github.com/Osmanyasal/Optimizer-Toolkit-Core.

[35] Candy Pang, Abram Hindle, Bram Adams, and Ahmed E. Hassan. What do programmers know about software energy consumption? *IEEE Software*, 33(3):83–89, 2016.

[36] https://icl.utk.edu/papi/.

[37] https://www.vi-hps.org/tools/tools.html.

[38] Hamid Saadatfar, Hamid Ahangar, and Javad Joloudari. A new dynamic game-based pricing model for cloud environment, 12 2023.

[39] Muhammad Aditya Sasongko, Milind Chabbi, Paul H J Kelly, and Didem Unat. Precise event sampling on amd versus intel: Quantitative and qualitative comparison. *IEEE Transactions on Parallel and Distributed Systems*, 34(5):1594–1608, 2023.

[40] Robert Schöne, Markus Velten, Daniel Hackenberg, and Thomas Ilsche. Energy efficiency features of the intel alder lake architecture. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*, ICPE '24, page 95–106, New York, NY, USA, 2024. Association for Computing Machinery.

[41] https://top500.org/lists/top500/2023/11/.

[42] Ondrej Vysocky, Martin Beseda, Lubomír Říha, Jan Zapletal, Michael Lysaght, and Venkatesh Kannan. Meric and radar generator: Tools for energy evaluation and runtime tuning of hpc applications: Third international conference, hpcse 2017, karolinka, czech republic, may 22–25, 2017, revised selected papers. pages 144–159, 07 2018.

[43] Bo Wang, Dirk Schmidl, Christian Terboven, and Matthias S. Müller. Dynamic application-aware power capping. In *Proceedings of the 5th International Workshop on Energy Efficient Supercomputing*, E2SC'17, New York, NY, USA, 2017. Association for Computing Machinery.