Power Scheduling for Maximizing Throughput and Fairness in Co-running Applications

SUNIL KUMAR, IIIT Delhi, India
VIVEK KUMAR, IIIT Delhi, India
SRIDUTT BHALACHANDRA, University of North Carolina at Chapel Hill, USA

Due to the significant cost associated with power consumption, hardware overprovisioning is widely used to cap processor power consumption and improve the average power utilization of servers in data centers and nodes in HPC clusters. However, uniformly capping power across sockets in multiprocessor servers can lead to performance degradation for co-running applications due to workload variability. Existing solutions primarily focus on cluster-level power management, making limited use of power scheduling within multi-socket servers or processor frequency scaling to regulate power consumption under power constraints.

This paper introduces Fulcrum, a novel power management library for co-running parallel applications on multi-socket, multi-core servers, independent of the underlying parallel programming model. Fulcrum dynamically redistributes power on a power-constrained multi-socket server to maximize throughput and fairness for co-running applications without requiring prior knowledge of application characteristics. Fulcrum periodically profiles hardware performance monitoring counters and independently adjusts core and uncore frequencies for each application based on its power sensitivity and degree of parallelism, thereby maximizing overall system throughput. After optimizing application-level power usage, Fulcrum enhances application-level fairness by dynamically redistributing power among applications, thereby maximizing aggregate throughput while adhering to the server's global power budget. We evaluated Fulcrum across various exascale proxy application mixes and power caps on a four-socket, 72-core Intel Cooper Lake processor. Our results show that Fulcrum improves system throughput (geometric mean) by 26.3% under low power caps and by up to 5.3% under higher power caps, while delivering power efficiency improvements of 27.7–8.4% at the respective power caps. Moreover, Fulcrum outperforms the two state-of-the-art approaches at both power caps, achieving geometric mean throughput improvements of 3.9–16.4% and power efficiency gains of 10.3–18.6%, while maintaining comparable fairness.

CCS Concepts: • **Hardware → Chip-level power issues**; • **Computer systems organization** → *Multicore architectures.*

Additional Key Words and Phrases: Power cap, DVFS, UFS, Multiprocessor Server, Throughput, Fairness

ACM Reference Format:
Sunil Kumar, Vivek Kumar, and Sridutt Bhalachandra. 2026. Power Scheduling for Maximizing Throughput and Fairness in Co-running Applications. 1, 1 (May 2026), 26 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

[0]New Paper, Not an Extension of a Conference Paper

Authors' Contact Information: Sunil Kumar, sunilk@iiitd.ac.in, IIIT Delhi, New Delhi, India; Vivek Kumar, vivekk@iiitd.ac.in, IIIT Delhi, New Delhi, India; Sridutt Bhalachandra, sriduttb@cs.unc.edu, University of North Carolina at Chapel Hill, Chapel Hill, NC, USA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM XXXX-XXXX/2026/5-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The number of compute elements, including cores and sockets, is increasing rapidly in modern multiprocessor servers used in cloud infrastructures, data centers, and supercomputers [2]. This advancement enables concurrent execution of multiple applications on multi-socket servers. However, studies have shown that the power consumption in data centers [28] and supercomputer sites [45] is significantly less than the provisioned power. The problem further exacerbates on systems that allow co-scheduling where disparity between the power demands of co-running applications and allocated power is the major cause of poor utilization. This advocates limiting power on such systems through *overprovisioning* in a manner that improves overall power utilization leading to higher system throughput while maintaining fairness across all jobs.

Each processor has a Thermal Design Power (TDP) rating from the manufacturer (measured in Watts). The processor TDP can be approximated as the sum of the TDPs of its components, such as core, uncore (chip components outside the CPU core), and integrated graphics (if any). By default, a processor's power consumption can approach its TDP while running compute-intensive kernels but typically remains lower while running complex workloads with interleaved compute and memory-bound phases. Power cap (PCAP) is a widely used technique in modern processors, used by data centers and supercomputing sites to enable hardware overprovisioning by restricting processor power consumption to levels below the TDP rating [44]. It allows reallocation of unused power to support additional computing resources, such as CPUs/GPUs or other components in the system. Modern processors support PCAP by throttling a processor's core and uncore frequencies at the hardware level. However, operating a processor under a PCAP can significantly degrade the performance for applications with high power demands (see Figure 1).

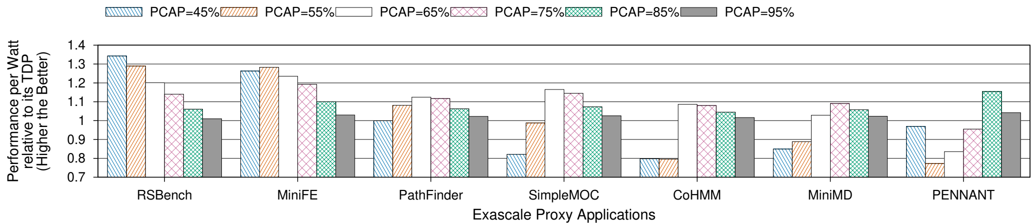


Fig. 1. Performance per Watt of ECP proxy applications on an Intel Cooper Lake processor across different PCAPs relative to TDP (PCAP shown as a percentage of TDP). Applications that improve performance per Watt at lower PCAPs demonstrate low power sensitivity, whereas those that degrade at lower PCAPs demonstrate high power sensitivity as they are more compute-intensive.

Several power management solutions exist for redistributing power across clusters to improve application performance [7, 10, 11, 19, 21, 31, 41, 43, 52, 62, 63]. These power managers (PMs) typically target cluster-wide scenarios such as idle phases caused by MPI barrier synchronization in a single application [7, 11, 21], dependencies in co-running applications due to in situ analysis or visualization [41, 62, 63], or sequential and I/O-bound phases in co-running independent applications [10, 43, 52]. However, there is a dearth of solutions focusing on power scheduling within a single multi-socket nodes while co-running applications. These scenarios of co-scheduling applications are becoming ever more prevalent as the compute densities of datacenter nodes increase.

An application's power usage could vary during its execution due to either a change in the Degree of Parallelism (DoP, the total number of active threads) or changes in its Memory Access Pattern (MAP). Power sensitivity reflects the performance impact an application experiences when

the PCAP changes, typically higher for compute-bound than memory-bound applications at the same DoP. Understanding the power sensitivity helps improve the system throughput of co-running applications by transferring power from the power-insensitive region of one application to the power-sensitive region of the other [19, 43]. However, due to variation in power sensitivity, several approaches rely on power prediction using ML models [19, 43, 62, 63]. Such models often lack scalability as they depend heavily on the target architecture or application behavior. In contrast, model-free power managers (PMs) offer greater scalability but lack awareness of the power sensitivity of co-running applications [10]. Solutions that can dynamically track the power sensitivity often lack the support for inter-socket power scheduling [25].

This paper introduces Fulcrum, a novel power management library for co-running applications on multi-socket servers independent of the underlying parallel programming model. It requires no application modifications or prior knowledge about the applications. Fulcrum runs a lightweight daemon that periodically profiles hardware Performance Monitoring Counters (PMUs) to dynamically detect each co-running application's MAP, DoP, and power usage. It employs the processor's core and uncore frequency scaling controls to dynamically classify the power sensitivity of the MAP of an application, and improve the performance of each co-runner within a user-defined PCAP leading to maximization of system throughput. Fulcrum uses Dynamic Voltage and Frequency Scaling (DVFS [35]) for core and Uncore Frequency Scaling (UFS [27]) for tuning the uncore. The uncore includes all chip components outside the CPU core [27], such as shared caches, memory controllers, and interconnects (UPI on Intel platforms). DVFS allows scaling down the core's voltage and frequency, thereby reducing power consumption as lowering the voltage has a squared effect on active power consumption. After optimizing application-level performance, Fulcrum dynamically redistributes power across the co-runners to improve system throughput, power efficiency and fairness, ensuring all co-running applications achieve comparable performance gains. To evaluate our all-inclusive solution (see Table 1), Fulcrum, we created several combinations of seven exascale proxy applications [3], using various parallel programming models.

In summary, this paper makes the following contributions:

- Fulcrum, a library-based power management framework for multi-socket multicore servers that improves fairness and throughput for co-running parallel applications under a system-wide power cap.
- A lightweight runtime profiler that monitors application's Degree of Parallelism (DoP) and Memory Access Pattern (MAP).
- A processor frequency throttle mechanism for dynamically detecting the power sensitivity of every MAP encountered during an application's execution and improving the performance under each MAP.
- A dynamic power redistribution mechanism that ensures application-level fairness by recalibrating application-level power budget, enabling comparable performance improvements across co-running applications.
- A comprehensive evaluation of Fulcrum against Slurm [60] and a recently published state-of-the-art implementation [10] on a quad-socket, 72-core Intel Xeon server using various exascale proxy application mixes and power caps, demonstrating significant gains in system throughput and energy savings.
- We open-source Fulcrum, along with the evaluated state of the art, the scripts, and the benchmarks used in this paper.[1].

System	Inter-socket Power Scheduling	ML Model Free	DoP Detection	Power Sensitivity	Fairness	DVFS	UFS
SLURM (JSSPP'03)~[60]	✓	✓	✓	✘	✘	✘	✘
PuPiL (ASPLOS'16)~[61]	✘	✓	✘	✘	✘	✓	✘
PTune (PACT'16)~[19]	✓	✘	✓	✓	✘	✘	✘
PERQ (HPDC'19)~[43]	✓	✘	✓	✓	✓	✘	✘
Guliani *etal*. (EuroSys'19)~[25]	✘	✓	✘	✓	✘	✓	✘
PoDD (SC'19)~[63]	✓	✘	✓	✘	✓	✘	✘
Bliss (PLDI'21)~[48]	✘	✘	✘	✓	✘	✓	✓
DPS (SC'23)~[10]	✓	✓	✓	✘	✓	✘	✘
Fulcrum	✓	✓	✓	✓	✓	✓	✓

Table 1. Comparison of state-of-the-art power management systems for co-running applications with Fulcrum

2 Related Work

Modern multicore processors provide multiple knobs for controlling power consumption, such as DVFS [35], UFS [27], and PCAP [38]. Prior research aimed at reducing the carbon footprint of modern supercomputers using a combination of these power-saving knobs. Broadly, this ongoing research seeks to improve a parallel application's energy efficiency at processor TDP [6, 11, 18, 20, 36, 47, 54–56] or its performance under a limited PCAP [15, 19, 21, 40, 43, 44]. Another approach for reducing the carbon footprint is to vary the number of active threads on a multicore processor to decrease power consumption [51, 53]. These approaches generally apply to applications that wait for synchronization or are constrained by memory.

Patki et al. studied the idea of hardware overprovisioning in a power-constrained HPC cluster and its impact on application performance in 2013 [44]. Recent studies have shown that power consumption in data centers [28] and supercomputer sites [45] is significantly lower than the provisioned power. Hardware *overprovisioning* can address the problem of power oversubscription and has been well explored at the cluster level since then [7, 10, 11, 15, 19, 21, 31, 40, 41, 44, 50, 52, 62, 63].

An early study proposed optimal distribution of power between the processor and DRAM [50], while others explored redistributing power across nodes to account for processor performance variation arising from synchronization wait time in MPI applications [7, 11, 19, 21, 40]. Node power usage during MPI synchronization barriers or collective communication is further reduced using concurrency throttling and DVFS [7, 40]. Another study considered performance variation in HPC clusters due to processor manufacturing variability and leveraged it to improve performance under power constraints [31]. PowerShift [62] demonstrated performance improvements for co-running applications at the cluster level under limited power. More recent studies [10, 52, 63] have proposed improvements over PowerShift by removing the need for offline analysis of application behavior.

Enhancing system throughput under power constraints (PCAP) at the multiprocessor server level has been widely studied [8, 23, 29, 30, 37, 58, 61, 65]. Pupil improves the performance of co-running applications using a combination of DVFS and concurrency throttling, but without power scheduling [61]. Zhu et al. [65] focus on efficiently partitioning compute resources between co-running applications on an integrated CPU-GPU architecture under a PCAP. Other solutions [8, 29, 58] apply processor frequency scaling but are restricted to single-application scenarios. In contrast, approaches such as [23, 30] adjust the PCAP dynamically rather than optimizing performance under a fixed PCAP. Bliss [48] uses Bayesian optimization to find the best configuration of DVFS, UFS, and thread count for single-application execution under PCAP, but lacks power scheduling support.

[1]Fulcrum code: https://doi.org/10.5281/zenodo.19475718

Benchmarks	Description	Memory Access Pattern	Configuration	Execution time (55%–100% of TDP)
CoHMM [16] (Kokkos)	Heterogeneous Multiscale Method	Xtreme Compute Bound (XCB)	Input=input3.txt, HM steps=2, MD steps=60	130sec-63sec
MiniMD [13] (Kokkos)	Parallel molecular dynamics		Input=in.lj.miniMD_comd, size=16x16x16, steps=27000	159sec-63sec
PathFinder [12] (OpenMP)	Signature search within graph	Compute Bound (CB)	Input=10kx1.5k.adj_list	257sec-115sec
SimpleMOC [26] (OpenMP)	3D neutron transport calculations		n_azimuthal=64, decomp_assemblies_ax=35	188sec-77sec
PENNANT [17] (OpenMP)	Unstructured mesh physics	Memory Bound (MB)	Input=leblancbig.pnt, tstop=11.0, mesh=120x1080	255sec-114sec
RSBench [57] (OpenMP)	Multipole resonance lookup		p=700000, P=1000000, W=100000	138sec-104sec
MiniFE [39] (Kokkos)	Finite Element Mini-Application	Xtreme Memory Bound (XMB)	nx=256, ny=512, nz=512	221sec-159sec

Table 2. Applications used for the evaluation of Fulcrum

Recent work [37] proposed a bidirectional power management runtime for co-running applications that dynamically redistributes power across sockets in a power-capped server. However, the runtime does not characterize application memory access patterns (MAPs) or perform core and uncore frequency optimization. Furthermore, its benefits are largely limited to scenarios where at least one co-runner experiences low-power utilization phases (LI region). Fulcrum also targets performance improvement for co-running parallel applications on multiprocessor systems, but uniquely identifies MAPs within applications and optimizes core and uncore frequencies for each MAP to improve overall system throughput. In addition, it improves application fairness through rebalancing of performance gains across co-runners. Similar to Fulcrum, Cuttlefish used TIPI to identify memory access patterns (MAPs) and optimize core and uncore frequencies for each MAP [36]. However, it was designed for improving energy efficiency at TDP and targets only single-application execution. Table 1 compares the capabilities of these runtime systems with those of Fulcrum.

3 Experimental Methodology

This section describes our experimental methodology before presenting our motivating analysis.

3.1 Applications

We used seven exascale proxy applications from the Exascale Computing Project (ECP) [3] for our experimental evaluations (see Table 2). ECP proxy application suite contains more than 80 applications. Our choice of applications is based on the following criteria: a) these applications could support multicore parallelism (OpenMP [42] and Kokkos [14]), b) they display a broad spectrum of DoP and MAP, and c) they cover a wide range of science domains and have been used in several prior studies [15, 19, 43, 48]. We have not modified these applications, except for porting CoHMM to Kokkos. The Kokkos's MiniMD and MiniFE implementations are already available in the ECP suite. We choose different parallel programming models in these applications to demonstrate the parallel runtime obliviousness in Fulcrum. Fulcrum is a parallel programming model–oblivious system and does not require any changes or runtime hooks within the application. The application binaries are provided to Fulcrum as command-line arguments to carry out profiling and power scheduling. Fulcrum uses the Unix *taskset* command to map application threads to the cores of a set of sockets and *numactl* to pin memory to the same set of sockets used by the application. We have used the standard settings of each application except the parameters highlighted in Table 2 to control the execution time on our machine.

3.2 Experimental Machine

We performed all experimental evaluations on a quad-socket Intel(R) Xeon(R) Gold Cooperlake 5318H processor, which has 18 cores per socket (totalling 72 cores). The turbo boost on system remains enabled by default. Our machine had 512GB of RAM and the Ubuntu 20.04.5 LTS operating system (OS) with Linux Kernel 5.4. The TDP of each socket was 150 Watts, resulting in a total system-level TDP of 600 Watts. We used `gcc/g++` compiler version 10.5 with `-O3` flag to compile our applications.

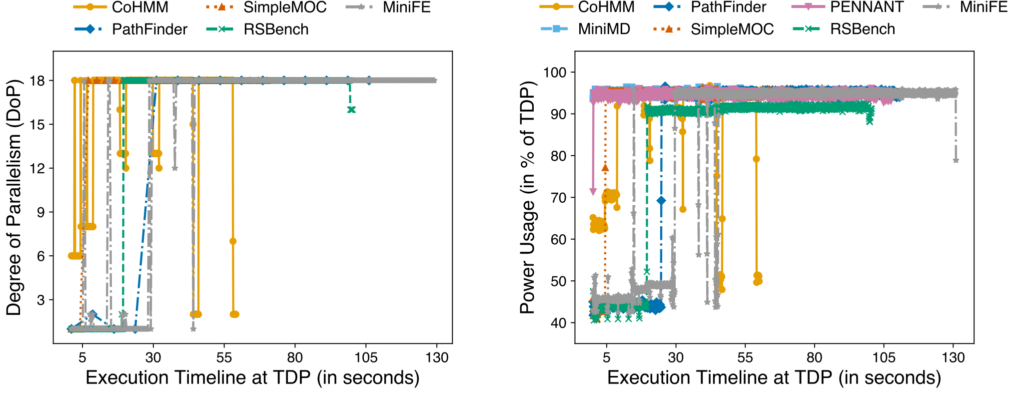
3.2.1 Power cap (PCAP). We used the Intel RAPL (Running Average Power Limit) interface [9] to set the socket-level PCAP by writing it into the `MSR_PKG_POWER_LIMIT` register of the socket. We set the short-term and long-term PCAP limits on the `MSR_PKG_POWER_LIMIT` register to the same value, ensuring power usage remains below the specified PCAP threshold. We also used RAPL's `MSR_PKG_ENERGY_STATUS` register for reading energy consumption (updated every 1 ms). AMD processors also provide a similar RAPL-supported interface for enabling PCAP.

3.2.2 Processor Frequency Scaling. The Linux kernel provides a range of power governor options via various frequency scaling drivers, including state-of-the-art power governors such as ondemand, conservative, and schedutils, which are supported by the `acpi-cpufreq` frequency scaling driver. In addition, Intel processors also support the `intel_pstate` driver, which holds the highest priority and is set as the default power governor on Intel machines [29]. The `intel_pstate` driver supports Intel Speed Shift Technology (`SST` [24]), a feature introduced with Xeon Skylake processors that allows for management of core and uncore frequency settings automatically at the hardware level, rather than through the Operating System (OS). `SST` is also referred to as Intel's *Hardware P-state policy* (HWP). We have adopted Intel HWP as the default power governor policy in all our experiments (including the baseline execution). Intel HWP on system remains enabled even while running Fulcrum policies that perform DVFS and UFS explorations. We are unaware of prior studies on optimizing DVFS and UFS carried over the Intel HWP power governor. Traditionally, the userspace power governor from `acpi-cpufreq` has been used for DVFS, and the same can also be used for DVFS explorations in Fulcrum. However, the limitation of `userspace` power governor is the lack of fine-grained control over P-state turbo frequency levels. In contrast, the `intel_pstate` driver enables more flexibility with DVFS by changing turbo levels within the predefined minimum and maximum turbo limits by directly editing the `scaling_min_freq` and `scaling_max_freq` files. Our experiments found that using Intel HWP during DVFS and UFS exploration provides more stable results than using `userspace` power governor from `acpi-cpufreq`. The DVFS frequency range supported in our system was 1.0GHz–3.8GHz (29 levels in total). The nominal frequency of our system was 2.5GHz. Our system's maximum supported turbo frequency is 3.8GHz, but the core frequency never goes beyond 3.3GHz when all cores are active due to thermal capping by system.

During Fulcrum execution, the uncore frequency is controlled dynamically by writing into the UFS MSRs `0x620` [22]. The UFS frequency range supported in our system was 1.1GHz–2.4GHz (14 levels in total). Our system supports the DVFS and UFS frequencies adjustments in steps of 100 MHz. The power scheduling approach in Fulcrum can be adapted to other processors, as discussed in Section 7.

4 Motivating Analysis

This section discusses how power usage varies with changes in a parallel application's Degree of Parallelism (DoP) and Memory Access Patterns (MAP). We used MAP to classify an application's power sensitivity. Additionally, we highlight the limitations of default core and uncore frequencies



(a) Applications transition between LI and LB regions due to fluctuations in DoP (DoP does not change with PCAP)

(b) Fluctuating power usage correlates with the changes in DoP

Fig. 2. Changes in DoP and power usage when running applications at the TDP

under a constrained power budget. Each experiment was conducted by running an application on a single socket (18 threads), with similar trends observed even when run on multiple sockets.

4.1 DoP and Power Variation

Figure 2(a) shows five applications from our chosen set that experience fluctuations in DoP throughout their execution at TDP (`CoHMM`, `MiniFE`, `PathFinder`, `RSBench`, and `SimpleMOC`). These applications encounter Load Imbalanced (LI) and Load Balanced (LB) regions. In the LI region, some threads remain idle, whereas in the LB region, all the threads remain active. The `CoHMM` application exhibits 28% LI region, during which the DoP fluctuates between 6, 8, and 13. On the other hand, `PathFinder`, `SimpleMOC`, and `RSBench` have LI regions of 22%, 5%, and 19%, respectively, where the DoP drops to one. Finally, `MiniFE` shows 36% LI region where the DoP is either 1 or 2. Figure 2(b) illustrates the power usage of each benchmark at the TDP, demonstrating that power consumption is lower during the LI regions. These power drops align with the decrease in DoP during the LI periods shown in Figure 2(a). The longer power drops in Figure 2(b) for `CoHMM`, `MiniFE`, `PathFinder`, and `RSBench` compared to `SimpleMOC` were due to the longer LI regions in these applications.

During LI regions, except `CoHMM`, each application's power consumption remained below the PCAP=45%, resulting in consistent LI durations for each application across higher PCAP levels. The power consumption of the LI region in `CoHMM` ranged between PCAP = 45–55%. The execution time of the LB region in each application increased as the PCAP decreased (see Table 2), primarily due to processor frequency throttling under the PCAP. LB regions in Compute-bound (CB) applications experienced a greater slowdown than memory-bound (MB) applications. However, due to the increasing execution time of the LB region with the decreasing PCAP level, the percentage of the LI region in an application decreases as the PCAP level decreases.

4.2 Variation in Memory Access Pattern

We used the metrics TOR (Table of Request) Inserts per Instruction (*TIPI*) [36] to classify the MAP[2] of the LB regions in each application. TIPI is calculated as the ratio of total (`TOR_INSERT.MISS_ALL`)

[2]We use the terms TIPI and MAP interchangeably throughout the paper.

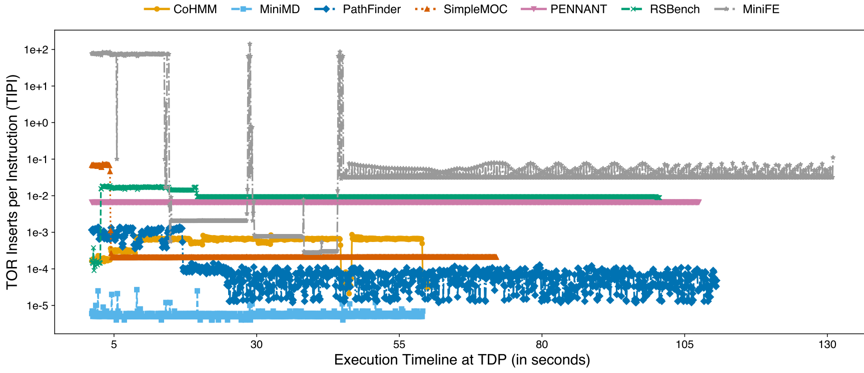


Fig. 3. Variation in TIPI during the application execution. Low TIPI implies compute-bound MAP and high TIPI implies memory-bound MAP (no change in TIPI at different PCAPs).

and total instructions retired (`INST_RETIRED.ANY`). Any requests coming to the LLC from the processor cores are placed in the TOR (Table of Request). `TOR_INSERT` MSR [32] is available on all Intel processors from Haswell generation and onwards. This uncore counter counts the number of memory requests that come to the socket-specific Last-Level Cache (LLC) from each core.

Figure 3 shows the TIPI of our applications throughout their execution. We can observe that the TIPI of an application fluctuates throughout its execution. The applications `PathFinder`, `RSBench`, and `SimpleMOC` are an exception as their TIPI overall remains consistent. Changing the PCAP or the processor's core and uncore frequencies does not affect an application's TIPI for the same problem size. We can classify the MAP of an application as CB and MB. The CB and MB applications have low and high TIPI values, respectively. As TIPI values fluctuate, we divided each unique TIPI into fixed slabs of 0.009 (derived empirically) to distinguish between unique TIPIs.

4.3 Relating TIPI with Power Sensitivity

In this section, we present an analysis that demonstrates how TIPI specifies the power sensitivity and the performance impact of the default core and uncore frequency set by Intel Hardware P-state policy for each TIPI. We executed each application by configuring three different pairs of core frequency (CF) and uncore frequency (UF) at PCAP=55%. To evaluate performance improvement for each TIPI, we measured total Instructions Retired (`INST_RETIRED.ANY`) per Second (IPS) [43]. We focused on prominent TIPIs, defined as those with occurrences greater than 5% of the total execution of LB region. The IPS values were normalized against the IPS for the corresponding TIPI under the default frequency settings (as dictated by the Intel Hardware P-state policy) at the same PCAP. The result of this experiment is shown in Figure 4 (similar trends were observed at other PCAPs). On the x-axis, we report the percentage of occurrences of the TIPIs during application execution, along with the processor set default frequencies. The processor could throttle frequencies if the user-set frequency needs more power than the user-set PCAP. The processor-throttled core and uncore (CF-UF) frequencies are reported above each bar.

We observed that the dominant TIPI range encountered in `CoHMM`, `MiniMD`, `PathFinder`, and `PENNANT` was between 0–0.009. By setting CF_{MAX} and UF_{MIN}, we achieved better performance for this MAP compared to the default frequency settings and the other two frequency combinations. This same TIPI range is also evident in `MiniFE`, where it constitutes 13% of its execution, maintaining a similar trend with respect to the frequency settings. The subsequent TIPI range (0.009–0.018) is

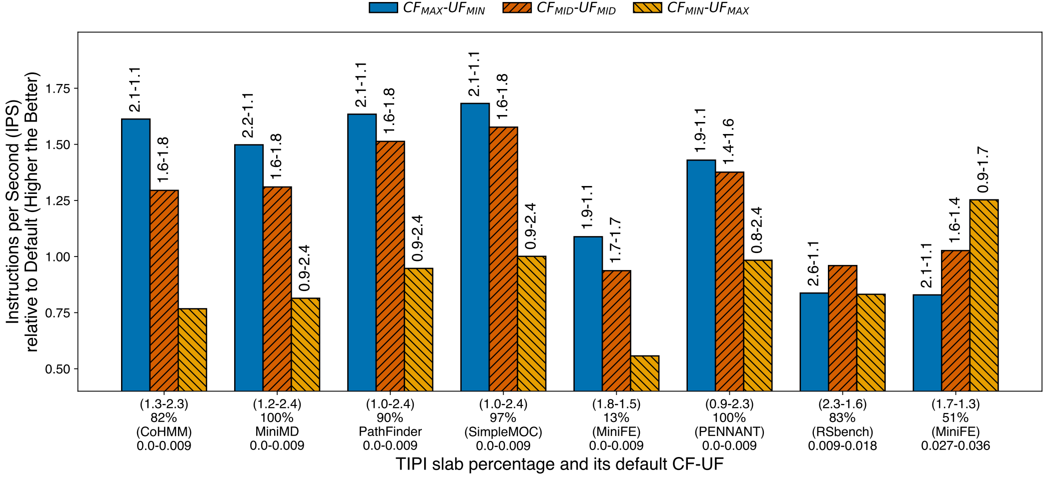


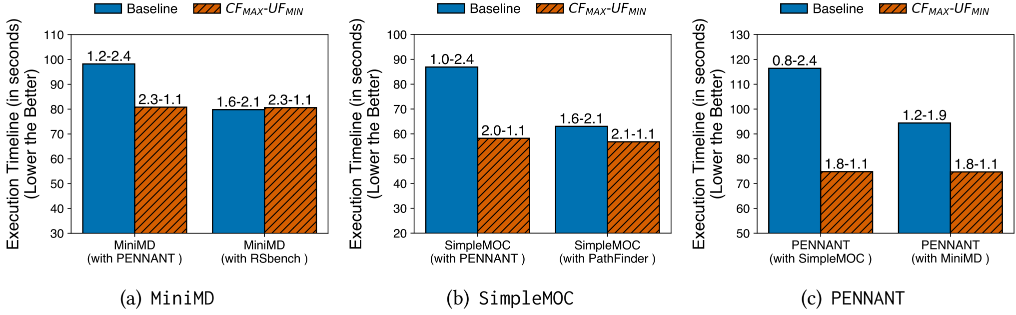
Fig. 4. IPS at different TIPIs normalized with the system default frequency settings at 55% PCAP. TIPI and its percentage of occurance is shown along with the stabilized frequencies (CF-UF) by the processor for achieving the power budget (similar trend at other three PCAPs)

prominent in RSBench, covering 83% of its execution, where CF_{MID} and UF_{MID} delivered superior performance for this MAP compared to the default and the other two frequency settings. Lastly, the dominant TIPI of MiniFE (0.027–0.036) exhibited better performance with CF_{MIN} and UF_{MAX} in comparison to the default and the other two frequency settings. From this results, we can observe that: a) low TIPI indicates a compute-bound (CB) MAP, whereas high TIPI indicates a memory-bound (MB) MAP, b) CB TIPIs have high power sensitivity than MB TIPIs, as they need higher core and lower uncore frequencies, and c) the processor default frequency settings are often suboptimal and there is a scope for improving the performance by exploring the optimal pair of core and uncore frequencies of unique TIPIs encountered during an application execution.

We have categorized the MAP of each of our applications into four distinct groups: XCB, CB, MB, and XMB, as illustrated in Table 2. This classification is based on the performance observed under various frequency settings, as shown in Figure 4. Due to a significant performance gap between the pairs CF_{MAX}–UF_{MIN} and CF_{MID}–UF_{MID}, we have classified CoHMM and MiniMD as XCB applications. The performance disparity between these two frequency pairs narrows for PathFinder and SimpleMOC, leading us to categorize these applications as CB. The CF_{MID}–UF_{MID} frequency setting performs similarly to or better than CF_{MAX}–UF_{MIN} in PENNANT and RSBench, resulting in their classification as MB. Lastly, MiniFE is classified as XMB as the CF_{MIN}–UF_{MAX} frequency settings outperform the other two settings.

4.4 Effect of Colocation on Execution time

The execution time of the same application can vary significantly depending on the co-running pair under a PCAP during baseline execution (Intel HWP). Figure 5 illustrates the results of this experiment for MiniMD, SimpleMOC, and PENNANT applications when paired with different applications. We also present the execution time of these applications with their respective pairs, using the core and uncore frequency combinations that delivered better performance, as shown in Figure 4. These optimal frequency pairs were only applied in the LB region of the respective applications (set of frequencies for each co-running application). The baseline execution times for



(a) MiniMD (b) SimpleMOC (c) PENNANT

Fig. 5. Execution time variation of MiniMD, SimpleMOC, and PENNANT with Intel HWP Policy at PCAP=55% under colocation with two different applications

MiniMD, SimpleMOC, and PENNANT differed with each pairing. However, by explicitly configuring the core and uncore frequencies according to the application's MAP, we achieved consistent execution times across different pairings. This observation further motivates the importance of optimal frequency explorations for applications running under PCAP.

4.5 Motivational Summary

Based on these motivational analyses, we can conclude that the execution of an HPC application consists of both LI and LB regions. In the LI region, power consumption remains low, while it varies based on the MAP in the LB region. LI regions can be identified through the DoP, while LB regions can be uniquely identified based on their TIPI, which also represents the power sensitivity. Additionally, within a constrained power budget, the core and uncore frequencies configured by Intel's Hardware P-state policy are often suboptimal and may lead to performance deterioration. Therefore, for co-running applications on a multi-socket processor, optimizing the frequencies on each socket and redistributing any unused power from one socket to another could enhance overall system throughput without overshooting the global power budget.

5 Design and Implementation

The previous section highlighted that the HPC application's power usage depends on the DoP and MAP, and the processor set frequencies under a limited PCAP could lead to performance degradation. Therefore, to enhance overall system throughput for co-running applications in a power-constrained multi-socket server, it is essential to dynamically optimize the core and uncore frequencies for each co-running application. It necessitates a low-overhead online profiler that could identify the MAP uniquely in each co-running application and explore the best combination of core and uncore frequencies for each MAP encountered in an application. The profiler should be able to detect low-power utilization phases in an application and dynamically schedule the unused power from one application to another. Further, to improve the application-level fairness, the profiler should assist in balancing the performance improvement of each application using bidirectional power scheduling.

To address the abovementioned challenges, we implemented a library-based runtime solution, Fulcrum, that employs a lightweight daemon process on a server for monitoring each co-running application's TIPI, IPS, DoP, and power usage without requiring application-level changes. The daemon activates itself at fixed intervals to minimize time-sharing overheads. Whenever the daemon discovers a new TIPI in any co-running application, it uses DVFS to explore the optimal core

frequency, followed by UFS to explore the optimal uncore frequency for that application's TIPI. An optimal frequency is the one that has the highest IPS within the available PCAP. Fulcrum daemon starts its execution without any information about the applications. Whenever an application enters an LI region (low DoP), the daemon distributes the unused power of this application to other applications until the DoP resets. When at least two applications are in the LB region, the daemon promotes fairness by balancing the performance improvement of these application using bidirectional power scheduling.

The following subsections discuss the design and implementation of Fulcrum using a running example of a mulitprocessor system with two sockets, $Socket_A$ and $Socket_B$. $PCAP_{user}$=83W (55% of TDP) is set on both these sockets ($PCAP_{user}$=166W at the system-level (Figure 6, 7). A single instance of the Fulcrum daemon process is launched on this system along with the applications App_A and App_B on $Socket_A$ and $Socket_B$, respectively. The daemon process is pinned to the last core on $Socket_A$, and the threads of App_A and App_B are pinned to the cores on their respective socket (the daemon process shares a core with the last thread in App_A). Applications do not share socket-level resources to assist Fulcrum in uniquely identifying TIPI and configuring optimal uncore frequency for each application, as both TIPI and UFS are applicable at the socket level. The `numactl` tool is used to ensure that the memory pages of each application are allocated on their socket-local DRAM.

5.1 Fulcrum daemon loop

Algorithm 1 lists the pseudocode implementation of the Fulcrum daemon process for a multi-socket multicore system. It sets the user-specified PCAP on each socket (Line 3). It runs in a loop as long as at least one application is running (Line 5). It then iterates over each co-running application (Line 7) by first measuring the application-level metrics (Line 8), followed by applying application-level Fulcrum runtime policy (Line 9–Line 49). Finally, it sleep for a fixed epoch of 100ms (Line 50). Due to cold caches, power readings fluctuate at the beginning of the application's execution (see in Figures 2(b)). Hence, to avoid recording unstable values for application-level metrics, the Fulcrum daemon loop activates only after a warmup duration of two seconds (Line 4).

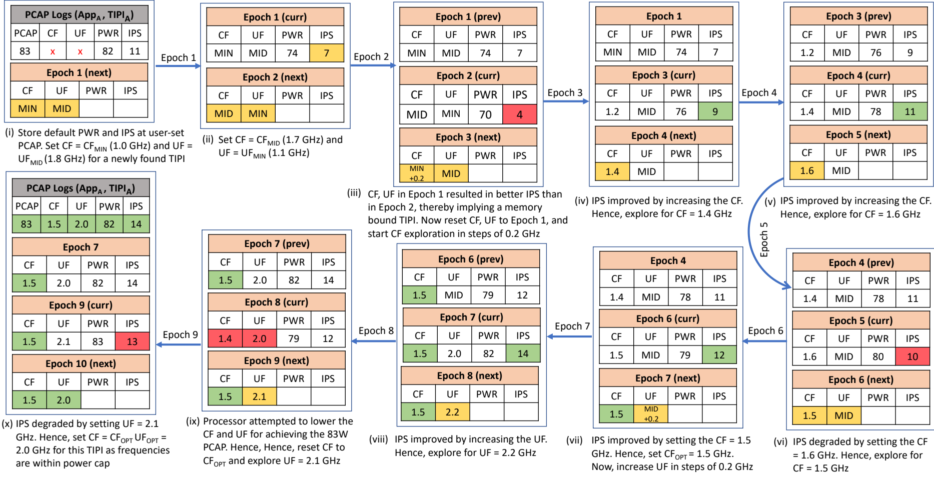
While executing the Fulcrum runtime policy (Line 9–Line 49) on App_A, the daemon will first check if the current TIPI of App_A is the same as in the previous epoch (Line 10). If either TIPI or DoP changes or App_A terminates, Fulcrum will reset the PCAP on App_A to the $PCAP_{user}$ (Line 31–Line 33). Otherwise, Fulcrum will classify the elapsed epoch on App_A into Load-Imbalanced (LI) region (Line 11) or Load-Balanced (LB) region (Line 13). It then explores the optimal core and uncore frequencies for App_A using the LI_Policy (Section 5.3) for the LI region (Line 12) or LB_Policy (Section 5.2) for the LB region (Line 14). After a few epochs running LI_Policy or LB_Policy, Fulcrum evaluates App_A power usage w.r.t. its allocated PCAP (Line 19). If there is unused power, then App_A is marked as the donor (Line 20). The surplus power is also recorded (Line 21), and the PCAP for App_A is reduced (Line 29). As App_B is the power recipient (Line 23), Fulcrum will increase the PCAP for App_B by adding the unused power from App_A to the $PCAP_{user}$ at App_B (Line 42 and Line 44). Fulcrum may require subsequent epochs to complete the frequency exploration under the increased PCAP on App_B. After that, it would start executing the Power Steering Policy (PS_Policy) at Line 17, as explained in Section 5.4. Meanwhile, App_B would continue executing at the newly set PCAP and its optimal frequencies. Whenever DoP or TIPI of App_A changes (Line 10), PCAP for App_A will be reset to $PCAP_{user}$ (Line 32). The core and uncore frequencies are also reset to the optimal values previously discovered at the $PCAP_{user}$. If each application is running an LB region whose LB_Policy has completed, and there is no unused power at either application, FC_Policy will be started (Line 48) for achieving application-level fairness (Section 5.5). FC_Policy redistributes the power across the co-running applications and then invokes PS_Policy for each application.

Algorithm 1: Fulcrum daemon thread method

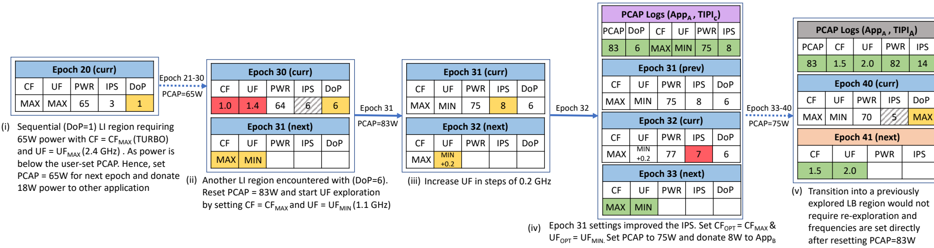
```
1  PCAP[N_Application] ⟵ {PCAP_user}
2  State_prev[N_Application] ⟵ {NEUTRAL}; PWR_Reset ⟵ 0
3  Set PCAP_user on each socket
4  sleep(warmup duration)
5  while atleast an Application is running do
6      PWR_extra ⟵ 0; N_Receiver ⟵ 0
7      for Application = 1 to N_Application do
8          Metrics ⟵ Read(TIPI, PWR, IPS, CF, UF, DoP)
9          State ⟵ NEUTRAL; isSet_LI/LB ⟵ isSet_PS ⟵ true
10         if DoP & TIPI remains unchanged then
11             if (DoP < Application_MaxDoP) then
12                 isSet_LI/LB ⟵ LI_Policy (Metrics)
13             else
14                 isSet_LI/LB ⟵ LB_Policy (Metrics)
15             end
16             if isSet_LI/LB & PCAP[Application]!=PCAP_user then
17                 isSet_PS ⟵ PS_Policy (Metrics)
18             end
19             if PWR<PCAP[Application] & isSet_LI/LB & isSet_PS then
20                 State ⟵ DONOR
21                 PWR_extra += PCAP[Application] - PWR
22             else
23                 State ⟵ RECEIVER; N_Receiver ++
24             end
25         end
26         if State != State_prev[Application]) then
27             PWR_Reset ⟵ 1
28             if State = DONOR then
29                 PCAP[Application] -= PWR
30             end
31             if State = NEUTRAL then
32                 PCAP[Application] = PCAP_user
33             end
34         end
35         State_prev[Application] ⟵ State
36     end
37     if PWR_Reset then
38         if PWR_extra > 0 then
39             PWR_extra = (N_Receiver>0) ? PWR_extra/N_Receiver:0
40             for Application = 1 to N_Application do
41                 if State = RECEIVER then
42                     PCAP[Application] += PWR_extra
43                 end
44                 Set PCAP[Application] on Application socket(s)
45             end
46             PWR_Reset ⟵ 0
47         else
48             PWR_Reset ⟵ (FC_Policy(Metrics) not done)
49         end
50     sleep ( T_epoch)
51 end
```

5.2 Load-Balanced Policy (LB_Policy)

LB_Policy is executed by the Fulcrum daemon process for exploring optimal core and uncore frequencies when an application is running with maximum DoP. Once the Fulcrum daemon loop activates, assuming it found App_A running an LB region. It will execute the LB_Policy for App_A for subsequent epochs until the optimal core and uncore frequencies are explored, as shown in Figure 6(a). This policy is executed for every unique and frequently found TIPI. CF and UF for App_A are either set to CF_{MID} and UF_{MIN} for a compute-bound TIPI or CF_{MIN} and UF_{MID} for a



(a) LB_Policy executing on App_A to determine the optimal core and uncore frequencies for $TIPI_A$



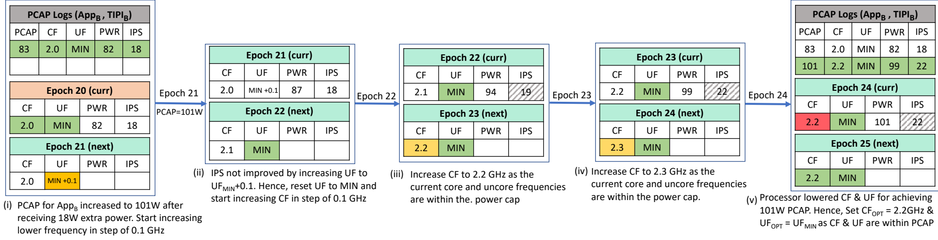
(b) LI_Policy executing on App_A to determine the optimal core and uncore frequencies in low-DoP regions

Fig. 6. Fulcrum's frequency throttling policies running on a dual-socket server with $PCAP_{user}$=83W at each socket (55% of TDP)

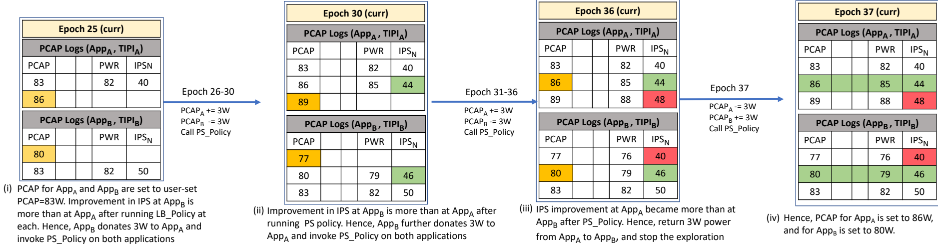
memory-bound TIPI. The frequency on the lower side is then explored for App_A until there is improvement in IPS (Figures 6(a)(iv)– 6(a)(vii)). Once the optimal value of the lower side frequency is found, exploration for the higher side frequency is carried out similarly (Figures 6(a)(iv)– 6(a)(x)). However, due to limited power budget, at some point, the processor could also attempt to lower the frequencies to achieve the fixed PCAP automatically (Figures 6(a)(ix)). Suppose there is a change in TIPI or region (LI). Fulcrum saves the exploration state for the previous TIPI (separately for each co-running application), resets the power to $PCAP_{user}$, and restarts the frequency exploration based on the region type.

5.3 Load-Imbalanced Policy (LI_Policy)

The Fulcrum daemon process executes LI_Policy to explore the optimal core and uncore frequencies when the DoP of an application drops. After a few epochs, assuming App_A transitioned into a LI region, the Fulcrum daemon would start executing the LI_Policy policy for App_A, as shown in Figure 6(b). The power usage will be minimal if the LI region uses a single core. Hence, Fulcrum will set CF to CF_{MAX} (turbo frequency) and UF to UF_{MAX} for this application (Figure 6(b)(i)). The processor would not attempt frequency throttling due to sequential execution in App_A. Due to the



(a) PS_Policy executing on App_B to improve IPS using surplus power received from App_A



(b) FC_Policy executing on App_A and App_B to achieve similar speedups through power redistribution

Fig. 7. Fulcrum's power transfer policies running on a dual-socket server with $PCAP_{user}$=83W at each socket (55% of TDP)

availability of unused power, App_A would now become a power donor. After a few epochs, App_A shift into a different LI region, now using half the total cores available on $Socket_A$ (Figure 6(b)(ii)). CoHMM is one such application that exhibits such a scenario. Fulcrum would restart LI_Policy for App_A by setting the CF to CF_{MAX} (turbo frequency) and UF to UF_{MIN} (Figure 6(b)(ii)). It will then attempt to explore the UF until there is improvement in IPS (Figures 6(b)(iii)– 6(b)(iv)). Fulcrum would save the optimal frequencies for this LI region for App_A according to the TIPI and DoP ((Figures 6(b)(iv)). App_A would continue to be a power donor but donate less power. Once App_A re-enters into a previously explored LB region ($TIPI_A$), Fulcrum will directly set the optimal frequencies for App_A (Figures 6(b)(v)).

5.4 Power Steering Policy (PS_Policy)

PS_Policy is shown in Figure 7(a), and is executed by the Fulcrum daemon process for an application when its PCAP increases after receiving power from a donor application. An application can become a power donor if running a LI region or due to power redistribution via the FC_Policy (Section 5.5). An application can become a power recipient only if it runs an LB region whose optimal frequencies have already been explored. This design simplifies the frequency exploration. If the power donor application has to donate power from an LB region, it would always keep some power as a buffer (10% of the current power usage) for itself and transfer the rest (not shown in Figure 7(a) for simplicity). This buffer is kept to avoid the processor's occasional throttling of the optimal frequencies. The goal of the PS_Policy is to increase the dominant frequency (core or uncore) for a power recipient application as high as possible under the increased PCAP. In our running example, App_A became a power donor of 18W ($PCAP_{user}$-$PCAP_{MIN}$) at epoch 20 by entering into a sequential LI region (Figures 6(b)(i)). App_B received the extra 18W from App_A in epoch 20. If App_B was executing an LB region running at optimal frequencies for $PCAP_{user}$, then Fulcrum will start running the PS_Policy

Mixes	Mix1 (OpenMP/Kokkos)	Mix2 (OpenMP-only)	Mix3 (OpenMP-only)	Mix4 (Kokkos-only)	Mix5 (OpenMP/Kokkos)	Mix6 (OpenMP/Kokkos)	Mix7
App0	CoHMM (XCB, Kokkos)	SimpleMOC (CB)	SimpleMOC	MiniMD (XCB)	MiniMD (Kokkos)	CoHMM (Kokkos)	MiniMD (Kokkos)
App1	PENNANT (MB)	PENNANT	PathFinder (CB)	MiniFE (XMB)	RSBench (MB)	PathFinder	SimpleMOC (MPI+OpenMP)
App2	-	-	-	-	-	-	RSBench (OpenMP)
App3	-	-	-	-	-	-	MiniFE (Kokkos)

Table 3. Details of the application mixes used for evaluations

for App_B as shown in Figure 7(a). Assume App_B was executing a compute-bound LB region (e.g., `SimpleMOC`) whose optimal frequencies were set in epoch 20, as shown in Figure 7(a)(i). PS_Policy will increase the PCAP to 101W for App_B and would then attempt first to raise the less dominant frequency (UF) until there is improvement in IPS (Figure 7(a)(i)). It will then attempt to increase the dominant frequency (CF) further as high as possible within the 101W PCAP (Figure 7(a)(ii)– 7(a)(iv)). PS_Policy sets the CF at turbo frequency for App_B for the entire duration of the sequential LI region in App_A (Figure 7(a)(v)), thereby improving the performance of App_B. Fulcrum also records the optimal frequency settings for App_B at PCAP=101W to avoid future explorations.

5.5 Fulcrum Policy (FC_Policy)

FC_Policy is executed by the Fulcrum to improve application-level fairness by carrying out dynamic load-balancing of power from one application to another. This policy is run once each co-running application runs in the LB region without any surplus power, and after LB_Policy has completed for each of these LB regions. Figure 7(b) shows the working of FC_Policy for App_A and App_B. Fulcrum measures the IPS improvement (IPS_N) by the LB_Policy for each application for their current TIPIs. The comparison is made with the IPS recorded for the same TIPI but before running the LB_Policy (i.e., when the processor set default frequencies was set). As IPS improvement for App_B is greater than that for App_A (threshold value is 5%), FC_Policy will reduce the PCAP for App_B by 3W and increase the PCAP for App_A by 3W (Figure 7(b)(i)). After updating the PCAPs, PS_Policy will be run for each application. FC_Policy will then reevaluate the IPS_N at each applications (Figure 7(b)(ii)). It will continue shifting 3W power from App_B to App_A, followed by running PS_Policy until either the IPS_N at each application is similar or until the IPS_N of App_A is more than that of App_B. As IPS improvement at App_A is now greater than that at App_B (see Figure 7(b)(iii)), FC_Policy will return 3W power to App_B from App_A and converge the exploration at this moment (see Figure 7(b)(iv).

6 Experimental Evaluation

This section presents the experimental evaluation of Fulcrum using the co-running mixes shown in Table 3. We created seven mixes of co-running applications, as shown in Table 3. The rationale behind these mixes was to create two and four co-running pairs, to combine applications with different degree of parallelism (DoP) behaviour and memory access patterns (MAPs), and include different parallel programming models (shown in the same Table). Each co-runner in each mix used two sockets (except Mix7) for its execution without sharing its socket local resources with the other co-runner (see Section 5). Each co-runner in Mix7 used a single socket. The SimpleMOC in Mix7 was executed with two MPI ranks, each rank using nine OpenMP threads with affinity set to the local socket. We performed evaluations at three different PCAPs: 83 Watts/socket (55% of TDP), 98 Watts/socket (65% of TDP), and 112 Watts/socket (75% of TDP). We executed each mix eight times and reported the mean value along with a 95% confidence interval.

Mixes		Mix1 (XCB-MB)		Mix2 (CB-MB)		Mix3 (CB-CB)		Mix4 (XCB-XMB)		Mix5 (XCB-MB)		Mix6 (XCB-CB)		Mix7 (XCB-CB-MB-XMB)			
PCAP	**Policy**	CoHMM	PENNANT	SimpleMOC	PENNANT	SimpleMOC	PathFinder	MiniMD	MiniFE	MiniMD	RSBench	CoHMM	PathFinder	MiniMD	SimpleMOC	RSBench	MiniFE
55%	FT_Policy	37.4 (±2.7)	63.9 (±1.7)	23.8 (±0.7)	62.4 (±1.9)	14.0 (±3.2)	2.6 (±0.3)	22.7 (±5.9)	7.3 (±0.8)	23.7 (±5.6)	1.2 (±0.5)	32.5 (±2.3)	6.7 (±1.5)	46.4 (±1.4)	46.6 (±2.5)	2.3 (±1.1)	11.9 (±0.9)
	Fulcrum	36.2 (±2.8)	71.3 (±1.2)	24.2 (±1.7)	66.4 (±2.1)	19.1 (±3.3)	12.0 (±0.5)	33.5 (±2.6)	8.6 (±1.4)	23.4 (±1.1)	4.5 (±0.2)	31.1 (±2.7)	16.8 (±1.5)	30.1 (±2.8)	28.8 (±2.5)	12.0 (±0.8)	13.8 (±0.9)
65%	FT_Policy	0.7 (±0.2)	54.4 (±0.7)	-1.5 (±1)	53.7 (±0.7)	-0.4 (±0.6)	-0.5 (±0.2)	9.0 (±1.9)	4.2 (±1.4)	9.9 (±1.5)	-0.8 (±0.4)	0.9 (±0.1)	-0.4 (±0.2)	12.5 (±1.1)	1.5 (±1.6)	1.6 (±1.0)	6.5 (±1.4)
	Fulcrum	5.6 (±0.1)	55.2 (±0.9)	10.6 (±0.9)	42.7 (±0.8)	5.4 (±0.8)	8.2 (±0.2)	17.6 (±0.9)	2.0 (±0.7)	9.9 (±0.8)	1.8 (±0.7)	4.6 (±0.2)	8.3 (±0.5)	12.4 (±1.3)	-1.1 (±2.8)	8.3 (±0.9)	6.6 (±0.8)
75%	FT_Policy	0.5 (±0.2)	26.7 (±0.5)	-3.3 (±1.9)	24.7 (±3.1)	-1.8 (±1.0)	-0.8 (±0.4)	0.5 (±2.7)	1.2 (±1.7)	1.4 (±0.9)	-0.5 (±0.6)	0.8 (±0.17)	-0.7 (±0.1)	-4.1 (±3.5)	-2.5 (±0.1)	-0.9 (±1.6)	0.3 (±1.8)
	Fulcrum	2.4 (±0.2)	25.9 (±0.5)	4.3 (±0.7)	21.9 (±0.5)	4.3 (±0.8)	4.8 (±0.2)	5.8 (±0.9)	-3.3 (±0.8)	3.5 (±0.7)	-0.6 (±0.5)	1.9 (±0.8)	3.3 (±0.6)	3.6 (±3.2)	2.5 (±3.9)	2.3 (±1.3)	0.2 (±0.9)

Table 4. Speedup (in %) of FT_Policy and Fulcrum over `Baseline` for individual applications in each mix across three PCAPs. Green and blue fonts indicate performance improvement and degradation, respectively, of Fulcrum relative to FT_Policy.

Optimal CF and UF settings for prominent TIPI Slabs using FT_Policy and `Baseline` (in GHz)																
Applications		CoHMM		MiniMD		PathFinder		SimpleMOC		PENNANT		RSBench		MiniFE			
TIPI Slab		0.00-0.008 XCB		0.00-0.008 XCB		0.00-0.008 CB		0.00-0.008 CB		0.00-0.008 MB		0.08-0.016 MB		0.00-0.008 XCB (11%)		0.024-0.032 XMB (54%)	
Mix Type		A	B	A	B	A	B	A	B	A	B	A	B	A	B	A	B
`Baseline`	**CF**	1.3	1.4	1.3	1.2	1.8	1.1	1.0	1.1	0.8	0.8	2.2	2.3	1.8	1.8	1.8	1.7
	UF	2.4	2.3	2.4	2.4	2.0	2.4	2.4	2.4	2.4	2.4	1.6	1.6	1.6	1.5	1.4	1.4
FT_Policy	**CF**	2.3	2.6	2.1	2.5	2.1	2.3	1.9	2.1	1.8	1.8	1.9	1.8	2.3	2.2	1.0	1.1
	UF	1.1	1.1	2.0	1.7	1.5	1.5	1.5	1.5	1.4	1.4	2.2	2.0	1.1	1.1	1.9	1.9

Table 5. Frequency Settings using FT_Policy at 55% PCAP for A and B mix types, where A denotes two-pair settings and B denotes four-pair settings

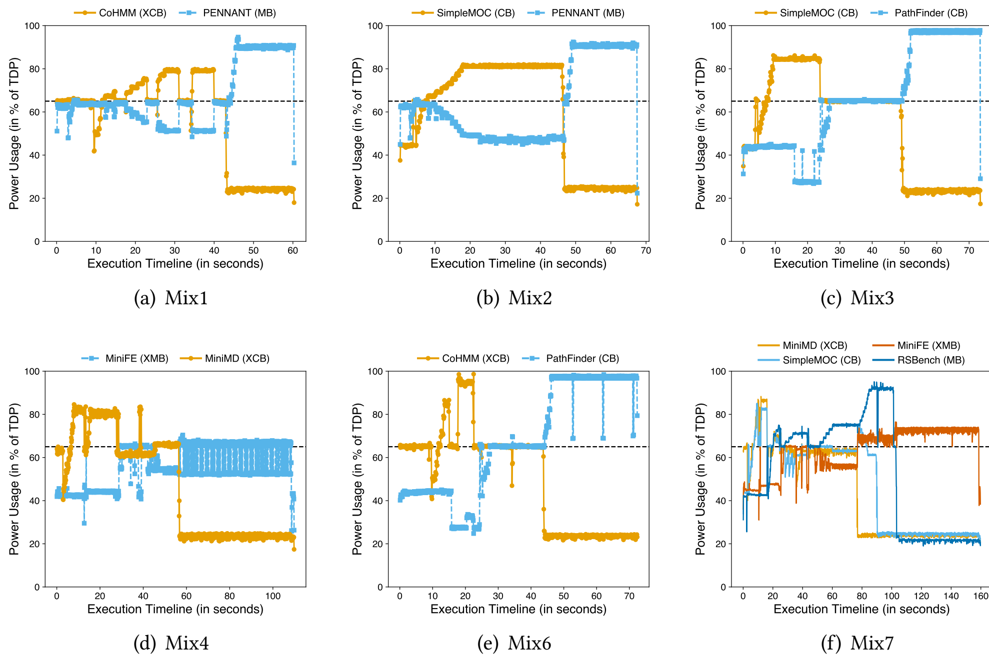
As described in Section 5, Fulcrum employs four runtime policies: LB_Policy, LI_Policy, PS_Policy, and FC_Policy. We refer to the combined use of LB_Policy and LI_Policy as FT_Policy (Frequency Throttling policy), whose evaluation is presented in Section 6.1. Fulcrum leverages all four policies to enhance application-level fairness, as shown in Section 6.2, and maximize overall system throughput and power efficiency, with results detailed in Section 6.3. Our `Baseline` execution does not apply any of the Fulcrum policies (see Section 3). Finally, in Section 6.4 we compare Fulcrum against the state-of-the-art implementations (Slurm [60] and DPS [10]). We chose these two state-of-the-art for experimental evaluation because they are also ML model–free and support inter-socket power scheduling and DoP detection similar to Fulcrum (see Table 1). While we evaluated Fulcrum on an Intel Cooperlake processor, it can be adapted for other processors as discussed in Section 7.

6.1 Speedup with FT_Policy under PCAP

Table 4 presents the speedups achieved by FT_Policy alone and by Fulcrum (using all four policies) over the `Baseline` execution for each application across different mixes and corresponding PCAP levels. Table 5 shows the core and uncore frequency (CF, UF) pairs set by FT_Policy and by the default processor behaviour under `Baseline` at 55% PCAP (similar trends were observed at other PCAP levels). The maximum performance gains achieved by `PENNANT`, `MiniMD`, `SimpleMOC`, `CoHMM`, `MiniFE`, `PathFinder`, and `RSBench` were 64.9%, 46.6%, 44%, 40.8%, 12.2%, 4.4%, and 2.1%, respectively. FT_Policy significantly improved application performance in cases where there was a notable discrepancy between the CF and UF pairs it selected and those chosen by the processor under `Baseline`. Such discrepancies arise in applications with TIPIs that fall clearly in compute-bound or memory-bound regions, such as `CoHMM`, `MiniMD`, `PathFinder`, `SimpleMOC`, and `MiniFE`. However, `PENNANT` and `RSBench` have TIPIs near the transition point between compute-bound and memory-bound regions. Although `PENNANT` is relatively more compute-bound than `RSBench`, the processor under `Baseline` set lower CF for `PENNANT` at lower PCAPs (see Figure 5). It allowed

FT_Policy to raise the CF and achieve significant performance gains (see Table 5). In contrast, RSBench, being more memory-bound, requires both high CF and UF for optimal performance. As the processor's default frequency pair under Baseline already closely matched that of FT_Policy, the gains from FT_Policy were minimal, with up to -1.3% performance loss at lower PCAPs, and -3.8% at higher PCAP. We also observed variability in SimpleMOC's default frequency settings across mixes at 55% PCAP (Mix2 and Mix3), resulting in different speedups achieved by FT_Policy. The performance improvements from FT_Policy reduce with increasing PCAP as the processor's default frequency settings are similar to those from FT_Policy. At 75% PCAP, some applications (SimpleMOC, PathFinder, and RSBench) exhibit performance loss with FT_Policy due to exploration overheads, as their performance gains from FT_Policy at lower PCAPs were already minimal. The performance improvement in MiniMD, MiniFE and SimpleMOC using FT_Policy was greater in the four-application (Mix7) than in the two-application mixes. This discrepancy is attributed to the higher degradation in Baseline owing to unoptimal frequency settings when an application is running on a single socket (see Table 5).

6.2 Speedup with Fulcrum under PCAP



(a) Mix1 (b) Mix2 (c) Mix3

(d) Mix4 (e) Mix6 (f) Mix7

Fig. 8. Power Steering across the co-running applications at PCAP of 65% of TDP

Figure 8 shows the timeline of power-steering from Fulcrum for four mixes at 65% PCAP. Fulcrum uses power steering to improve the system throughput and application fairness (Section 5). The reduction in power at the sockets of one application increases the power of the other sockets. A similar trend holds for all other mixes. The Fulcrum results reported in Table 4 demonstrate that it either improved the fairness by degrading the performance (highlighted in blue font) of one co-runner and enhancing the performance of the other (highlighted in green font), or improved

the performance of both the co-runners (both co-runner's performance highlighted in green font in such cases). The latter was due to the performance improvement from PS_Policy during the LI regions (even due to early termination of any co-runner).

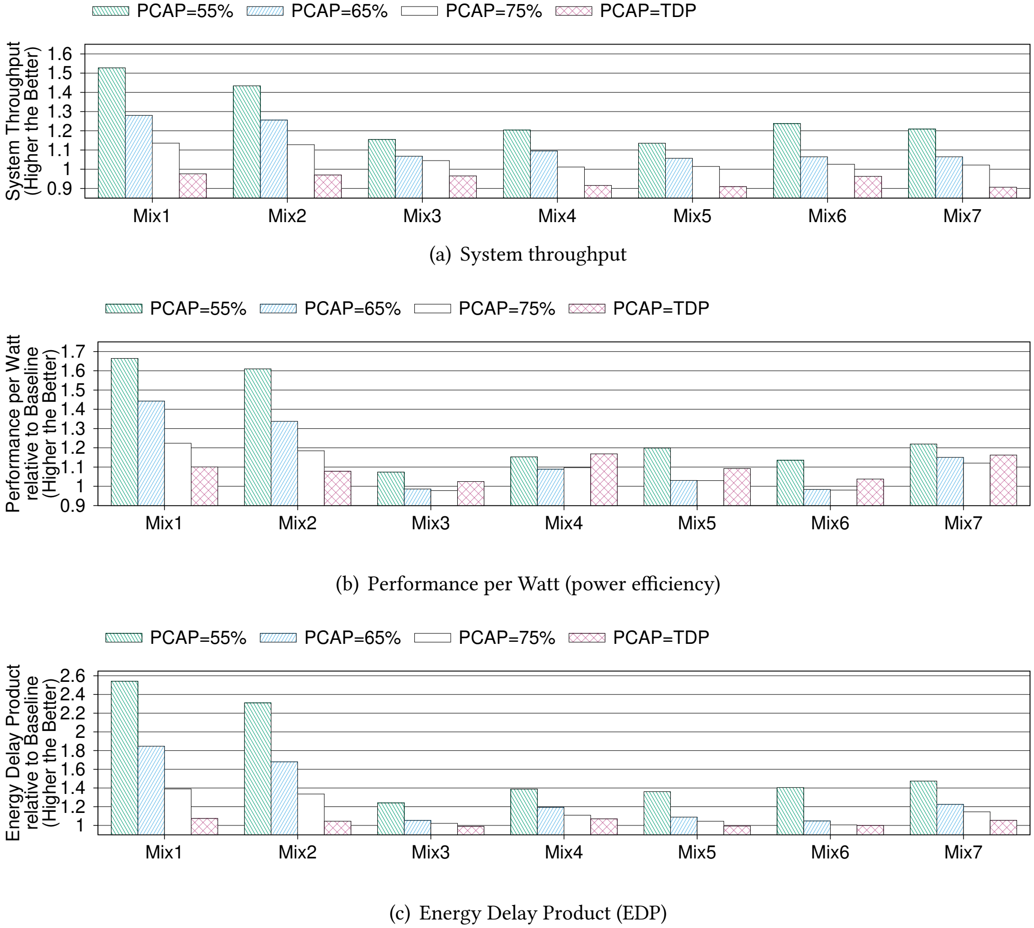
Fulcrum improved the performance of the low-performing applications in each mix by 2.5%–10.5% over FT_Policy (Mix7). The fairness policy in Fulcrum becomes effective whenever any co-runner enters the LB region in any mix. Application-level fairness is achieved by turning low-performing applications into power receivers and others into power donors (Figure 8(a) and Figure 8(b)). Mix3 is a unique case where Fulcrum improved system throughput at all PCAP levels but did not enhance application fairness. It is because the improvement in the IPS of the LB region from LB_Policy was similar for both `SimpleMOC` and `PathFinder` in Mix3 due to their comparable TIPIs. Consequently, the performance gains in `SimpleMOC` under Fulcrum were driven mainly by LB_Policy and the surplus power available from the LI region in `PathFinder`, ranging from 15% to 45% of TDP across all PCAPs (Figure 8(c)).

`PENNANT` in Mix2 (PCAP=65%) and `SimpleMOC` and `MiniMD` in Mix7 (at PCAP=55%) experienced significant performance degradation due to the fairness policy. Each of these three applications is highly power sensitive, leading to significant degradation at these lower PCAPs. FC_Policy continually transfers power from `PENNANT` to `SimpleMOC` and from both `SimpleMOC` and `MiniMD` to `RSBench` in Mix2 and Mix7, respectively. We can notice that in Mix2 and Mix7, `SimpleMOC` and `RSBench` achieved 12.3% and 10.5% performance gain, respectively, demonstrating the efficacy of FC_Policy (see Figures 8(b) and 8(f)). As the PCAP approaches the TDP, however, the frequencies set by FT_Policy converge with the processor's default frequencies, diminishing the impact of the fairness policy in all mixes.

6.3 System throughput and Power Efficiency using Fulcrum

Figure 9(a) presents the overall system throughput achieved by Fulcrum across all mixes and PCAP levels (including TDP). Throughput improvements over the `Baseline` execution were calculated using the geometric mean formula [49] shown in Figure 10. We used Performance per Watt as the metric to demonstrate the power efficiency [34]. As discussed in Section 6.2, performance gains from Fulcrum are marginal at higher PCAPs due to the near-optimal frequency settings in the `Baseline` execution. From Figure 9(a), we observe that at 75% PCAP levels and at TDP, Fulcrum performs similarly to `Baseline` for all mixes except Mix1, Mix2, Mix3, and Mix7. These four mixes demonstrate higher throughput across all PCAPs except TDP. This improvement in throughput ranges between 4.8% (Mix3 at PCAP=75%) and 52.6% (Mix1 at PCAP=55%). It was primarily due to the presence of `SimpleMOC`, `PENNANT`, and `MiniMD`, which receive a substantial performance boost from FT_Policy (see Table 4). This boost allows Fulcrum to enhance co-runner performance by donating power from these applications. Although Mix4, Mix5, and Mix6 perform similarly to `Baseline` at PCAP=75%, these mixes demonstrated significant improvement in system throughput at PCAP=55% and 65%, ranging between 5.7–9.5% (PCAP=65%) to 13.5–22.3% (PCAP=55%). Mix4 and Mix5 contain the least power-sensitive application (`RSBench` in Mix5 and `MiniFE` in Mix4), whereas Mix6 has overlapping LI regions between `CoHMM` and `PathFinder` (see Figure 8(e)).

We can observe from Figure 9(b) that Fulcrum was unable to achieve better throughput at TDP. We included throughput comparison of Fulcrum for each mix at TDP, to demonstrate its applicability even when not having any power constraints. This result also demonstrates the overheads of using Fulcrum (overhead ranging between 12-2.3% across all mixes and PCAPs). Although none of the applications are power efficient at TDP (Figure 1), Fulcrum still improves the power efficiency even at TDP. Fulcrum was able to significantly improve power efficiency across all mixes at other PCAP levels (except Mix3 and Mix6). In Mix3 and Mix6, the power efficiency was improved only at 55% and 65% PCAPs (ranging between 7.2–12.2%). These two mixes delivered lower power efficiency at



(a) System throughput

(b) Performance per Watt (power efficiency)

(c) Energy Delay Product (EDP)

Fig. 9. Overall improvement in system throughput, performance per watt (power efficiency), and EDP relative to Baseline by Fulcrum for all mixes at each PCAP and TDP

$$\text{Throughput}_{\text{Mix}} = \left(\prod_{i=1}^{N} s_{\text{i}} \right)^{\frac{1}{N}}$$

- $\text{Throughput}_{\text{Mix}}$: Geometric mean of all s_{i}
- N: Number of applications used in Mix
- s_{i}: Speedup of the policy over the Baseline execution for i-th application

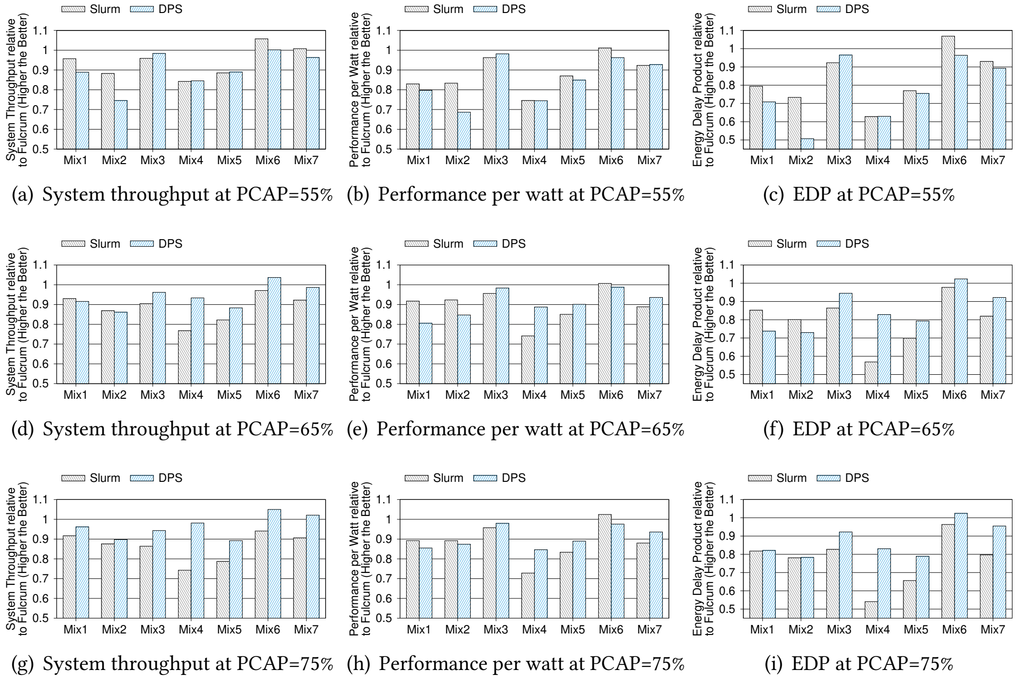
Fig. 10. Formula for calculating overall system throughput

PCAP=75% as they contain two out of CoHMM, PathFinder, or SimpleMOC. These three applications are least power sensitive, even though they are compute-bound. From Figure 1 we can observe that these applications demonstrated peak power efficiency at 65% PCAP, thereby mixes containing these applications show the least power efficiency beyond 65% PCAP.

Energy–Delay Product (EDP) is a widely used metric for evaluating the trade-off between performance and energy efficiency in computing systems. It is defined as the product of total energy consumption and execution time. Figure 9(c) presents the EDP achieved by Fulcrum across all

mixes and PCAPs relative to `Baseline`. We observed that the overall improvement in EDP across all mixes reached up to 61.4% at PCAP=55%, 27.4% at PCAP=65%, 14.2% at PCAP=75%, and 3.2% at TDP, demonstrating the efficacy of Fulcrum across all PCAP settings. Moreover, unlike system throughput and performance per watt, Fulcrum is able to improve EDP for each mix even at TDP.

6.4 Comparison of Fulcrum with State-of-the-Art



(a) System throughput at PCAP=55% (b) Performance per watt at PCAP=55% (c) EDP at PCAP=55%

(d) System throughput at PCAP=65% (e) Performance per watt at PCAP=65% (f) EDP at PCAP=65%

(g) System throughput at PCAP=75% (h) Performance per watt at PCAP=75% (i) EDP at PCAP=75%

Fig. 11. Overall improvement in system throughput, performance per watt, and EDP by Slurm and DPS relative to Fulcrum for all mixes

Figure 11 compares the system throughput and performance per watt achieved by Slurm [60] and DPS [10] relative to Fulcrum across all mixes and three PCAP levels. We ported DPS from its original cluster-level design to support socket-level power management on a single server and implemented a single-server version of Slurm following its documentation [1]. The functional differences among Slurm, DPS, and Fulcrum are summarized in Table 1. DPS maintains a history of power usage for co-running applications and adapts power allocation accordingly, whereas Slurm is a stateless power manager that reacts only to instantaneous power usage. Hence, as observed in Figure 11, between Slurm and DPS, Slurm achieves better throughput at lower PCAP (55%), whereas DPS achieves better throughput at higher PCAPs (65% and 75%). Because DPS is history-aware, its adaptive power allocation becomes more effective when power is less constrained. At low PCAPs, Slurm can respond more quickly, as it does not need to maintain a power usage history resulting from the extended power profiling process in DPS.

Overall, Fulcrum consistently outperformed both Slurm and DPS across all mixes and PCAPs, except for Mix6 and Mix7. In Mix7, Fulcrum performed similarly to or better than both Slurm and DPS, whereas in Mix6, Slurm or DPS performed better. Slurm achieved 5.7% higher throughput

than Fulcrum in Mix6 at PCAP=55%, whereas DPS achieved up to 5.0% higher throughput than Fulcrum at higher PCAPs in Mix6. This was due to exploration overheads in Fulcrum resulting from frequent transitions between LI phases of `CoHMM` and `PathFinder` (see Figure 8(e)). Across all mixes, the throughput improvement of Fulcrum relative to DPS decreased as PCAP increased. This is due to the diminishing benefit of FT_Policy with increasing PCAP. DPS trailed Fulcrum in throughput marginally by up to 5.7% in Mix3, as Fulcrum primarily benefited from power transfer from the sequential to the parallel region using PS_Policy. In contrast, Fulcrum achieved significant throughput improvements relative to both Slurm and DPS in Mix1, Mix2, Mix4, and Mix5, as each mixes contains a power-sensitive application, such as `PENNANT` in Mix1 and Mix2 and `MiniMD` in Mix4 and Mix5. Both `PENNANT` and `MiniMD` demonstrated significant performance improvements with FT_Policy (see Table 4). A similar trend is observed for power efficiency (Figures 11(b), 11(e), and 11(h)) and EDP (Figures 11(c), 11(f), and 11(i)).

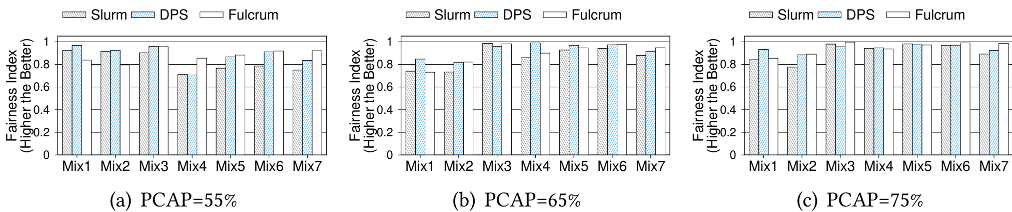
6.5 Fairness achieved by Fulcrum

Figure 13 presents the fairness achieved by Fulcrum, Slurm, and DPS across all mixes and PCAPs. The fairness index is computed using a variant of Jain's fairness index [33] (see Figure 12). The original formulation is expressed as $\frac{1}{1+\text{CoV}^2}$. Specifically, the fairness value is computed as $1 - \text{CoV}$, where CoV is the cofficient of variance between speedups achieved by each application in a given mix under a specific runtime implementation [49]. When the fairness value approaches one, it indicates perfect fairness, i.e., the applications in a mix achieve similar speedup gains. It can also be negative, indicating severe imbalance among applications. This fairness metric helps capture small performance imbalances that are often not reflected by Jain's index as it suppresses outliers and ignores minor imbalances.

$$\text{Fairness}_{\text{Mix}} = 1 - \frac{\sigma_s}{\mu_s}, \quad \text{where } \sigma_s = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(s_i - \mu_s)^2}, \quad \mu_s = \frac{1}{N}\sum_{i=1}^{N} s_i.$$

- $\text{Fairness}_{\text{Mix}}$: $1 - \text{CoV}$, where $\text{CoV} = \frac{\sigma_s}{\mu_s}$
- N: Number of applications used in Mix
- s_i: Speedup of the policy over the `Baseline` execution for i-th application
- μ_s: Mean speedup across all applications
- σ_s: Standard deviation of speedup values

Fig. 12. Formula for calculating fairness among co-running applications



(a) PCAP=55% (b) PCAP=65% (c) PCAP=75%

Fig. 13. Fairness achieved by Fulcrum for all mixes and at each PCAPs

From Figure 13, we observe that, except for Mix1 across all PCAPs and for Mix2 at PCAP = 55%, Fulcrum performs similarly to or better than both Slurm and DPS. Additionally, Fulcrum provides higher fairness at higher PCAPs than at PCAP = 55%. `PENNANT` achieves significantly higher speedup with FT_Policy than with `CoHMM` and `SimpleMOC` in Mix1 and Mix2, respectively

MSR Profiling overhead (in %)	PCAP=55%	PCAP=65%	PCAP=75%
Min	1.0	0.8	0.7
Max	5.7	1.5	1.1

Table 6. MSR profiling overhead in Fulcrum daemon process across all mixes

Settling Time (in %)	PCAP=55%	PCAP=65%	PCAP=75%
Min	0.4	1.6	2.0
Max	1.8	4.7	6.4

Table 7. Settling time for finding optimal core and uncore frequencies by FT_Policy across all mixes

(see Table 4). As a result, even after applying FC_Policy, disparities remain in the speedup gains between the applications in these mixes. In general, the FT_Policy improves performance for all applications at low PCAPs due to suboptimal frequency settings chosen by the processor in the baseline execution. Additionally, each mix contains a pair of applications with different power sensitivities (except Mix3), which causes disparities in speedup gains across applications. Owing to the similar power sensitivities of the applications in Mix3, we observed the highest fairness provided by Fulcrum among all mixes at PCAP = 55%.

6.6 Overheads of Fulcrum

In this section, we discuss the runtime overheads associated with Fulcrum. We first present the profiling overheads (Section 6.6.1), followed by frequency exploration overheads (Section 6.6.2), and the overheads associated with selecting TIPI slab size (Section 6.6.3).

6.6.1 Profiler overhead. The minimum and maximum overheads of MSR profiling at each PCAP are shown in Table 6. We calculated this overhead by first running each mix with the Fulcrum daemon and then running it without the daemon, and reporting the difference in execution time as the profiling overhead. When using the Fulcrum daemon, we only performed MSR profiling in this experiment and did not execute any Fulcrum policies. We observed that, except for a 5.7% overhead at PCAP = 55%, the overheads were otherwise always below 2%. The overhead decreased as the PCAP increased. It was slightly higher at PCAP = 55% because the core frequencies are near their minimum, which affects core sharing between the application thread and the profiling daemon. The profiler overhead does not change as the number of co-running applications increases, as it depends on the total number of available sockets on a server rather than the number of running applications.

6.6.2 Settling time. We used the settling time metric [61] to quantify the overhead associated with exploring the optimal core and uncore frequencies. It is defined as the total duration spent by the Fulcrum daemon in determining the optimal core and uncore frequencies for different TIPIs encountered during benchmark execution. Considering that the total number of DVFS and UFS levels in our system was 43 and the profiling epoch was 100 ms, in the worst case, the total time spent by FT_Policy would be 4.3 s per TIPI. However, Fulcrum never encounters this worst-case scenario due to the optimizations used to determine the frequency exploration ranges (see Section 5.2). Table 7 presents the minimum and maximum settling times observed across all mixes at each PCAP. We found that the settling time across all mixes and PCAPs ranges between 0.4% and 6.4%. The settling time increases with increasing PCAP due to the wider range of available frequencies for exploration. At lower PCAPs, the exploration range is smaller, resulting in shorter settling times.

6.6.3 Frequent phase change in applications. Each TIPI represents a unique phase during an application's execution. We chose a default TIPI slab size of 0.009 for all experiments reported in this paper. Selecting a finer-grained TIPI slab size increases the total number of unique TIPIs identified during an application's execution, thereby simulating fast phase-changing behavior. As shown in

	TIPI Slab Size		
	0.006	0.003	0.001
Average performance relative to TIPI Slab of 0.009	-0.3%	-1.1%	-1.2%
Unique TIPIs identified	15	19	38
Average settling time (in %)	3.5%	6.8%	7.7%

Table 8. Impact of TIPI slab size granularity on MiniFE

Figure 3, among all the applications used in this paper, MiniFE exhibits the widest range of TIPI fluctuations. Therefore, we executed Mix4 (which includes MiniFE) while varying the granularity of the TIPI slab sizes. Table 8 presents the results of this experiment. A coarser-grained TIPI slab size reduced the number of TIPIs identified and thereby decreased the settling time in MiniFE, whereas a finer-grained TIPI slab size increased the number of TIPIs identified, thereby increasing the settling time due to frequent oscillations in frequency settings caused by rapid phase changes. However, the overall overhead introduced by these frequency oscillations was limited to 7.7% even after reducing the TIPI slab size by 9×. For the default TIPI slab size of 0.009 used in this paper, the total number of TIPIs identified in MiniFE was 8, and the average speedup across all three PCAPs was 4.2% (see Table 4). In future work, we plan to dynamically determine and configure the optimal TIPI slab size for each application rather than using a fixed slab size.

7 Discussion

We evaluated Fulcrum on an Intel Cooper Lake processor, but it can be ported to other processors. Intel has provided support for RAPL MSRs since the Sandy Bridge architecture, and support for UFS and uncore-related PMUs since the Haswell architecture. Therefore, deploying Fulcrum on other Intel processors primarily requires updating the MSRs and the supported frequency ranges for DVFS/UFS. Enabling DVFS, UFS, and PCAP requires one-time BIOS-level configuration changes. AMD EPYC processors provide support for DVFS. Recent studies report that AMD EPYC processors from Zen3 onward support PMUs for uncore components such as Infinity Fabric and the Unified Memory Controller (UMC) [46, 64]. The Infinity Fabric clock (IFCLK) supports multiple frequency levels that can be adjusted at the OS level through AMD's HSMP interface [4]. PCAP support is available through a RAPL-compatible interface exposed in Linux on recent AMD platforms. Although we used both DVFS and UFS in our evaluation, Fulcrum can also operate on systems where UFS is not supported. Modern ARM processors expose PMUs for monitoring uncore components such as the interconnect and memory subsystem [59], and provide runtime DVFS and power management capabilities via firmware interfaces such as the ARM SCMI protocol [5].

8 Conclusion

This paper proposes a programming model oblivious power management runtime for improving the system throughput, energy efficiency, and application-level fairness for co-running HPC applications in a power-constrained multi-socket server. The runtime enhances the performance of each co-runner by adapting the application-specific processor frequencies using DVFS and UFS according to the instantaneous power usage of each application. Any unused power by an application is distributed across the busy applications to improve the server throughput and energy efficiency. When each application fully utilises its allocated power, the runtime improves the application-level fairness by redistributing the power across the co-runners so that each co-runner achieves similar performance benefits. In the future, we will extend the runtime for a CPU-GPU server and use it at the cluster level.

9 Acknowledgement

This research was supported by the Google PhD Fellowship 2022 program.

References

[1] [n. d.]. SchedMD. https://slurm.schedmd.com/SLUG15/Power_mgmt.pdf
[2] November 2025. TOP500. https://top500.org/lists/top500/2025/11/
[3] Released. *ECP Proxy Applications*. https://proxyapps.exascaleproject.org/ecp-proxy-apps-suite/
[4] AMD. 2025. AMD HSMP. https://github.com/amd/amd_hsmp
[5] Arm Limited. [n. d.]. *Arm® System Control and Management Interface (SCMI) Specification*. Arm Limited. https://developer.arm.com/architectures/system-architectures/software-standards/scmi
[6] Solomon Abera Bekele, M Balakrishnan, and Anshul Kumar. 2019. ML Guided Energy-Performance Trade-Off Estimation For Uncore Frequency Scaling. In *2019 Spring Simulation Conference (SpringSim)*. 1–12. doi:10.23919/SpringSim.2019.8732878
[7] Sridutt Bhalachandra, Allan Porterfield, Stephen L. Olivier, and Jan F. Prins. 2017. An Adaptive Core-Specific Runtime for Energy Efficiency. In *IPDPS '17*. 947–956. doi:10.1109/IPDPS.2017.114
[8] Luis Costero, Francisco D. Igual, and Katzalin Olcoz. 2023. Dynamic power budget redistribution under a power cap on multi-application environments. *Sustainable Computing: Informatics and Systems* 38 (2023), 100865. doi:10.1016/j.suscom.2023.100865
[9] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. 2010. RAPL: memory power estimation and capping. In *ISLPED '10* (Austin, Texas, USA). Association for Computing Machinery, New York, NY, USA, 189–194. doi:10.1145/1840845.1840883
[10] Jianru Ding and Henry Hoffmann. 2023. DPS: Adaptive Power Management for Overprovisioned Systems. In *SC '23*. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3581784.3607091
[11] Jonathan Eastep, Steve Sylvester, Christopher Cantalupo, Brad Geltz, Federico Ardanaz, Asma Al-Rawi, Kelly Livingston, Fuat Keceli, Matthias Maiterth, and Siddhartha Jana. 2017. Global extensible open power manager: a vehicle for HPC community collaboration on co-designed energy management solutions. In *ISC '17*. Springer, Cham, 394–412. doi:10.1007/978-3-319-58667-0_21
[12] ECP-copa. v1 release. *PathFinder Graph Search Mini-App*. https://github.com/Mantevo/PathFinder
[13] ECP-copa. v2.0 release. *A simple proxy for the force computations in typical molecular dynamics applications*. https://github.com/Mantevo/miniMD
[14] H Carter Edwards, Christian R Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *JPDC '14* (2014), 3202–3216. doi:10.1016/j.jpdc.2014.07.003
[15] Daniel A. Ellsworth, Allen D. Malony, Barry Rountree, and Martin Schulz. 2015. Dynamic Power Sharing for Higher Job Throughput. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*. Article 80, 11 pages. doi:10.1145/2807591.2807643
[16] ExMatEx. commit id d8776bf. *Co-design proxy application for the Heterogeneous Multiscale Method*. https://github.com/exmatex/CoHMM/tree/sad
[17] Charles R. Ferenbaugh. v0.9 release. *Unstructured mesh hydrodynamics for advanced architectures*. https://github.com/lanl/PENNANT
[18] R. Ge, Xizhou Feng, and K.W. Cameron. 2005. Performance-constrained Distributed DVS Scheduling for Scientific Applications on Power-aware Clusters. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*. 34–34. doi:10.1109/SC.2005.57
[19] Neha Gholkar, Frank Mueller, and Barry Rountree. 2016. Power Tuning HPC Jobs on Power-Constrained Systems. In *PACT '16*. 179–191. doi:10.1145/2967938.2967961
[20] Neha Gholkar, Frank Mueller, and Barry Rountree. 2019. Uncore Power Scavenger: A Runtime for Uncore Power Conservation on HPC Systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19)*. Article 27, 23 pages. doi:10.1145/3295500.3356150
[21] Neha Gholkar, Frank Mueller, Barry Rountree, and Aniruddha Marathe. 2018. PShifter: Feedback-Based Dynamic Power Shifting within HPC Jobs for Performance. In *HPDC '18 (HPDC '18)*. 106–117. doi:10.1145/3208040.3208047
[22] Corey Gough, Ian Steiner, and Winston Saunders. 2015. *Energy efficient servers: blueprints for data center optimization*. Springer Nature.
[23] Amina Guermouche. 2022. Combining uncore frequency and dynamic power capping to improve power savings. In *IPDPSW '22*. IEEE, 1028–1037. doi:10.1109/IPDPSW55747.2022.00164
[24] Part Guide. 2011. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: system programming guide, Part* 2, 11 (2011), 1–64.

[25] Akhil Guliani and Michael M Swift. 2019. Per-application power delivery. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16. doi:10.1145/3302424.3303981

[26] Geoffrey Gunow and John Tramm. v4 release. *Performance Characterterics and Viability of the Method of Characteristics (MOC) for 3D neutron transport calculations*. https://github.com/ANL-CESAR/SimpleMOC

[27] David L Hill, Derek Bachand, Selim Bilgin, Robert Greiner, Per Hammarlund, Thomas Huff, Steve Kulick, and Robert Safranek. 2010. The Uncore: A Modular Approach to Feeding the High Performance Cores. *Intel Technology Journal* 14, 3 (2010).

[28] Chang-Hong Hsu, Qingyuan Deng, Jason Mars, and Lingjia Tang. 2018. SmoothOperator: Reducing Power Fragmentation and Improving Power Utilization in Large-scale Datacenters. In *ASPLOS'18*. 535–548. doi:10.1145/3173162.3173190

[29] Darong Huang, Luis Costero, and David Atienza. 2024. Is the powersave governor really saving power?. In *CCGrid '24*. https://doi.org/10.1109/CCGrid59990.2024.00039

[30] Connor Imes, Huazhe Zhang, Kevin Zhao, and Henry Hoffmann. 2019. Copper: Soft real-time application performance using hardware power capping. In *ICAC '19*. IEEE, 31–41. doi:10.1109/ICAC.2019.00015

[31] Yuichi Inadomi, Tapasya Patki, Koji Inoue, Mutsumi Aoyagi, Barry Rountree, Martin Schulz, David Lowenthal, Yasutaka Wada, Keiichiro Fukazawa, Masatsugu Ueda, et al. 2015. Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing. In *SC '15*. 1–12. doi:10.1145/2807591.2807638

[32] Intel. Accessed 2021. Intel Xeon processor E5 v3 family uncore performance monitoring. https://www.intel.com/content/dam/www/public/us/en/zip/xeon-e5-v3-uncore-performance-monitoring.zip

[33] Raj Jain, Arjan Durresi, and Gojko Babic. 1999. Throughput fairness index: An explanation. In *ATM Forum contribution*, Vol. 99. 1–13.

[34] Shoaib Kamil, John Shalf, and Erich Strohmaier. 2008. Power efficiency in high performance computing. In *2008 IEEE International Symposium on Parallel and Distributed Processing*. 1–8. doi:10.1109/IPDPS.2008.4536223

[35] Wonyoung Kim, Meeta S. Gupta, Gu-Yeon Wei, and David Brooks. 2008. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *HPCA '08*. 123–134. doi:10.1109/HPCA.2008.4658633

[36] Sunil Kumar, Akshat Gupta, Vivek Kumar, and Sridutt Bhalachandra. 2021. Cuttlefish: Library for Achieving Energy Efficiency in Multicore Parallel Programs. In *SC '21*. Article 81, 14 pages. doi:10.1145/3458817.3476163

[37] Sunil Kumar and Vivek Kumar. 2025. KarmaPM: Reward-Driven Power Manager. In *European Conference on Parallel Processing*. Springer, 351–365. doi:10.1007/978-3-031-99854-6_24

[38] Charles Lefurgy, Xiaorui Wang, and Malcolm Ware. 2008. Power capping: a prelude to power shifting. *Cluster Computing* 11 (2008), 183–195. doi:10.1007/s10586-007-0045-4

[39] Mantevo. v2.2.0 release. *Finite Element Mini-Application*. https://github.com/Mantevo/miniFE

[40] Aniruddha Marathe, Peter E Bailey, David K Lowenthal, Barry Rountree, Martin Schulz, and Bronis R de Supinski. 2015. A run-time system for power-constrained HPC applications. In *High Performance Computing: 30th International Conference, ISC High Performance 2015, Frankfurt, Germany, July 12-16, 2015, Proceedings 30*. Springer, 394–408. doi:10.1007/978-3-319-20119-1_28

[41] Ivana Marincic, Venkatram Vishwanath, and Henry Hoffmann. 2020. SeeSAw: Optimizing Performance of In-Situ Analytics Applications under Power Constraints. In *IPDPS '20*. IEEE, 789–798. https://doi.org/10.1109/IPDPS47924.2020.00086

[42] OpenMP ARB. November 2018. *OpenMP Application Programming Interface Version 5.0*. https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[43] Tirthak Patel and Devesh Tiwari. 2019. PERQ: Fair and Efficient Power Management of Power-Constrained Large-Scale Computing Systems. In *HPDC '19*. 171–182. doi:10.1145/3307681.3326607

[44] Tapasya Patki, David K. Lowenthal, Barry Rountree, Martin Schulz, and Bronis R. de Supinski. 2013. Exploring Hardware Overprovisioning in Power-Constrained, High Performance Computing. In *ICS '13*. 173–182. doi:10.1145/2464996.2465009

[45] Tapasya Patki, Barry Rountree, Torsten Wilde, Andrea Bartolini, Stephanie Brink, Esa Heiskanen, Sachin Idgunji, Matthias Maiterth, James Rogers, Ermal Rrapaj, Ralf Schneider, Woong Shin, Kathleen Shoga, Christian Simmendinger, Nicholas J. Wright, and Zhengji Zhao. 2025. A Global Perspective on Supercomputer Power Provisioning: Case Studies from United States and Europe. In *ICS '25*. 1034–1051. https://doi.org/10.1145/3721145.3734532

[46] Xu Peter. 2025. Tuning UEFI Settings for Performance and Energy Efficiency on 5th Gen AMD EPYC Processor-Based ThinkSystem Servers. https://lenovopress.lenovo.com/lp2210.pdf

[47] Barry Rountree, David K. Lowenthal, Bronis R. de Supinski, Martin Schulz, Vincent W. Freeh, and Tyler Bletsch. 2009. Adagio: Making DVS Practical for Complex HPC Applications. In *Proceedings of the 23rd International Conference on Supercomputing (ICS '09)*. ACM NY USA, 460–469. doi:10.1145/1542275.1542340

[48] Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, and Devesh Tiwari. 2021. Bliss: Auto-Tuning Complex Applications Using a Pool of Diverse Lightweight Learning Models. In *PLDI '21*. 1280–1295. doi:10.1145/3453483.3454109

[49] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2021. SATORI: Efficient and Fair Resource Partitioning by Sacrificing Short-Term Benefits for Long-Term Gains. In *ISCA '21*. 292–305. https://doi.org/10.1109/ISCA52012.2021.00031

[50] Osman Sarood, Akhil Langer, Laxmikant Kalé, Barry Rountree, and Bronis de Supinski. 2013. Optimizing power allocation to CPU and memory subsystems in overprovisioned HPC systems. In *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. 1–8. doi:10.1109/CLUSTER.2013.6702684

[51] Srinath Sridharan, Gagan Gupta, and Gurindar S. Sohi. 2014. Adaptive, Efficient, Parallel Execution of Parallel Programs. In *PLDI '14*. ACM, 169–180. doi:10.1145/2594291.2594292

[52] Tapan Srivastava, Huazhe Zhang, and Henry Hoffmann. 2023. Penelope: Peer-to-peer Power Management. In *ICPP '22* (Bordeaux, France). doi:10.1145/3545008.3545047

[53] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. 2008. Feedback-Driven Threading: Power-Efficient and High-Performance Execution of Multi-Threaded Workloads on CMPs. In *ASPLOS '08*. ACM, 277–286. doi:10.1145/1346281.1346317

[54] Vaibhav Sundriyal and Masha Sosonkina. 2011. Per-call Energy Saving Strategies in All-to-All Communications. In *Recent Advances in the Message Passing Interface*, Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra (Eds.). Springer Berlin Heidelberg, 188–197. doi:10.1007/978-3-642-24449-0_22

[55] Vaibhav Sundriyal, Masha Sosonkina, Bryce M. Westheimer, and Mark Gordon. 2018. Comparisons of Core and Uncore Frequency Scaling Modes in Quantum Chemistry Application GAMESS. In *Proceedings of the High Performance Computing Symposium (HPC '18)*. Society for Computer Simulation International, Article 13, 11 pages.

[56] Ananta Tiwari, Michael Laurenzano, Joshua Peraza, Laura Carrington, and Allan Snavely. 2012. Green Queue: Customized Large-Scale Clock Frequency Scaling. In *2012 Second International Conference on Cloud and Green Computing*. 260–267. doi:10.1109/CGC.2012.62

[57] John R. Tramm, Andrew R. Siegel, Benoit Forget, and Colin Josey. v13 release. *Performance Analysis of a Reduced Data Movement Algorithm for Neutron Cross Section Data in Monte Carlo Simulations*. https://github.com/ANL-CESAR/RSBench

[58] Bo Wang, Julian Miller, Christian Terboven, and Matthias Müller. 2020. Operation-aware power capping. In *Euro-Par '20*. https://doi.org/10.1007/978-3-030-57675-2_5

[59] Yun Wu, Dimitrios S. Nikolopoulos, and Roger Woods. 2016. Runtime support for adaptive power capping on heterogeneous SoCs. In *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*. 71–78. doi:10.1109/SAMOS.2016.7818333

[60] Andy B Yoo, Morris A Jette, and Mark Grondona. 2003. Slurm: Simple linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*. Springer, 44–60.

[61] Huazhe Zhang and Henry Hoffmann. 2016. Maximizing Performance Under a Power Cap: A Comparison of Hardware, Software, and Hybrid Techniques. In *ASPLOS '16* (USA). 545–559. doi:10.1145/2872362.2872375

[62] Huazhe Zhang and Henry Hoffmann. 2018. Performance & Energy Tradeoffs for Dependent Distributed Applications Under System-wide Power Caps. In *ICPP '18* (Eugene, OR, USA). doi:10.1145/3225058.3225098

[63] Huazhe Zhang and Henry Hoffmann. 2019. PoDD: Power-capping dependent distributed applications. In *SC '19*. 1–23. doi:10.1145/3295500.3356174

[64] Zhong Zheng, Seyfal Sultanov, Michael E Papka, and Zhiling Lan. 2025. Minimizing Power Waste in Heterogenous Computing via Adaptive Uncore Scaling. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 505–518. doi:10.1145/3712285.3759879

[65] Qi Zhu, Bo Wu, Xipeng Shen, Li Shen, and Zhiying Wang. 2017. Co-run scheduling with power cap on integrated cpu-gpu systems. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 967–977. doi:10.1109/IPDPS.2017.124