Energy-Aware Runtime Resource Harmonizer for Co-running Applications

Vanshika Jain
IIIT Delhi
India

Varun Parashar
IIIT Delhi
India

Vivek Kumar
IIIT Delhi
India

Chiranjib Sur
Shell India Markets Pvt. Ltd, India
Krea University, India

***Abstract*—Nowadays, due to the increasing number of cores and sockets in modern multiprocessor servers, it is essential to co-schedule multiple applications simultaneously to maximize system utilization. However, the performance and energy efficiency of co-executing applications are highly sensitive to thread placement, core allocation, and both core and uncore frequency settings. Existing dynamic resource allocation solutions often rely on model-based approaches, require intrusive modifications to the parallel runtime, or lack a unified framework that can utilize multiple energy-saving mechanisms while supporting different parallel programming models.**

This paper presents *Harmonizer*, a novel dynamic resource optimization library for co-running applications on multiprocessor systems to improve overall system throughput and energy efficiency. Harmonizer is oblivious to the parallel programming model and requires no information from prior executions. It periodically profiles each application's CPU, cache, and memory usage by utilizing hardware performance monitoring counters and uses this data to determine optimal thread placement, core allocation, and frequency settings. We evaluate Harmonizer using several co-running mixes of exascale proxy applications on a four-socket, 72-core Intel Cooper Lake processor. Our results show that Harmonizer reduces energy consumption by 8.8% to 35% (20.5% geometric mean) and improves throughput by 4.8% (geometric mean) compared to the default Linux scheduler. Compared to two state-of-the-art approaches, it achieves up to 28.6% energy savings and 15.8% higher throughput (geometric mean).

***Index Terms*—multiprocessor systems, co-running, resource allocation, thread placement, DVFS, UFS, DCT, energy efficiency**

I. Introduction

The number of compute elements, including cores and sockets, is increasing rapidly in modern multiprocessor systems used in cloud infrastructures, data centres, and supercomputers [1]. This advancement has enabled concurrent execution of multiple applications on multiprocessor servers [2], [3], [4]. However, simultaneously achieving high performance and energy efficiency in co-running applications remains a significant challenge. Each application has unique and dynamic resource requirements that change throughout execution [5]. Parallel applications typically alternate between load-imbalanced and load-balanced phases, which influence overall CPU utilization. CPU-bound applications generally require more cores than memory-bound ones due to their higher computational demands [6]. Dynamic Concurrency Throttling (DCT) addresses this by dynamically adjusting core allocation through thread packing and unpacking, thereby optimizing energy efficiency based on real-time CPU utilization [7]. In addition, modern processors offer fine-grained control over core and uncore frequencies via Dynamic Voltage and Frequency Scaling (DVFS) and Uncore Frequency Scaling (UFS), providing further opportunities to optimize energy efficiency [8], [9]. DVFS enables scaling down a core's voltage and frequency, thereby reducing power consumption, as active power is proportional to the square of the voltage. UFS operates at the socket level, controlling the frequency of uncore components such as the last-level cache (LLC), memory controllers, and interconnects. However, resource sharing at the uncore level presents significant challenges. Shared last-level cache (LLC) can degrade performance when one application evicts data required by another, while contention at the memory controller can create bottlenecks as applications compete for memory bandwidth. These issues are further exacerbated by the increasing number of sockets per node, where non-uniform memory access (NUMA) introduces additional latency between local and remote memory regions [5].

TABLE I
Comparison of state-of-the-art resource management frameworks for co-running HPC applications with Harmonizer

System	Runtime Agnostic	Model Free	Thread Placement	DVFS	UFS	DCT
NuPoCo (PACT'18 [6])	×	✓	✓	×	×	✓
Abera et al. (TACO'21 [9])	×	×	×	✓	✓	×
MAPPER (TACO'22 [2])	✓	✓	✓	×	×	✓
Eflect (ICSE'22 [10])	✓	✓	×	✓	×	×
FCUFS (CLUSTER'24 [11])	×	×	×	✓	✓	×
Harmonizer	✓	✓	✓	✓	✓	✓

Various approaches have been proposed to efficiently manage co-running applications on multi-socket, multicore systems by dynamically allocating runtime resources such as cores, cache, and memory bandwidth. Preemptive scheduling improves application grouping [12], [13], [14], [15], but its effectiveness depends on the presence of multiple queued jobs. While prioritizing specific application mixes can help reduce contention, it may also result in degraded overall system throughput. Moreover, preemptive scheduling typically requires changes to the underlying operating system [14], which affect all applications and users on a server, making such solutions difficult to adopt in practice. Hardware resource partitioning enhances dynamic resource utilization [16], [3], but often leverages learning-based models to minimize exploration overhead and reduce dependence on specific hardware architectures. Many solutions utilize hardware Performance Monitoring Counters (PMCs) for lightweight profiling, en-

abling dynamic core allocation and thread mapping [6], [17], [2], [18]. Others extend this approach to optimize cache and memory bandwidth allocation [4]. Energy-efficient techniques based on processor frequency scaling typically target isolated applications [19], [8], [20], [21] or employ learning-based strategies for co-running workloads [9], [11].

Among the strategies mentioned above, PMC-based approaches offer the most flexible and portable means of resource optimization for co-running applications. However, existing methods lack a unified framework that can adaptively balance the conflicting objectives of high performance and low energy consumption. In particular, current solutions do not jointly optimize thread placement, per-application core allocation, and both core and uncore frequency scaling in a coordinated manner. Since UFS operates at the socket level, co-running applications with different uncore access patterns on the same socket pose tuning challenges, limiting energy-saving potential. Therefore, it is essential to refine thread placement strategies to ensure that applications with similar runtime characteristics are mapped to the same socket. This paper addresses these challenges by introducing Harmonizer, a comprehensive dynamic resource optimization framework for co-running parallel applications on multiprocessor systems, designed to enhance both system utilization and energy efficiency. Table I compares the capabilities of Harmonizer with the state-of-the-art.

In summary, this paper makes the following contributions:

- Harmonizer, a library-based framework for optimizing resource utilization in multi-socket, multicore servers, independent of parallel runtime systems, concurrency decomposition strategies, and prior knowledge of application behaviour.
- A lightweight daemon process that runs alongside co-running applications, periodically profiling CPU, cache, and memory usage using PMCs.
- A dynamic core redistribution policy that adjusts core allocation based on each application's instantaneous CPU usage.
- Thread placement policies to reduce cache contention, optimize memory bandwidth usage, and facilitate optimal socket-level uncore frequency tuning.
- Coordinated use of DVFS and UFS to dynamically tune core and uncore frequencies at each socket, improving overall energy efficiency.
- Experimental evaluation of Harmonizer on a modern multiprocessor system using nine mixes of Exascale Computing Project (ECP) proxy applications [22], demonstrating substantial improvements in both system throughput and energy efficiency.

II. Related Work

Efficient resource utilization techniques for co-running applications on multi-socket, multicore systems have been widely studied. Prior work on userspace-based solutions has explored a variety of strategies to address these challenges specifically for non-preemptive co-running applications, as summarized below.

Hardware resource partitioning techniques, such as Intel Cache Allocation Technology (CAT) and Memory Bandwidth Allocation (MBA) [23], are widely used to determine optimal partitions for last-level cache (LLC) and memory bandwidth, respectively. Dicer [24] leverages CAT to partition the LLC based on individual application's cache usage. However, it is designed to optimize the partitioning of a single shared resource (LLC) among co-running applications. In contrast, CLITE [25] and SATORI [16] apply Bayesian optimization to simultaneously optimize multiple resources, including LLC (via CAT), memory bandwidth (via MBA), and core allocation. DRLPart [26] and JointOpt [27] employ offline model training to make online partitioning decisions. A standard limitation of these approaches is their dependence on model-based optimization, which often results in limited scalability due to tight coupling with specific architectures or application behaviours. PARTIES [28] avoids model-based techniques and instead employs a feedback-driven mechanism to dynamically optimize shared resources, including cache partitioning, core allocation, core frequency, and disk bandwidth. Similarly, OLPart [3] uses hardware performance counters to guide the dynamic partitioning of cores, cache, and memory bandwidth. Unlike SATORI, implementations such as Dicer, CLITE, PARTIES and OLPart were designed for microservices, which are typically loosely coupled and latency-sensitive, rather than for HPC applications that are tightly coupled, throughput-oriented, and often exhibit highly CPU-intensive parallel phases. SATORI targeted HPC applications, but it was designed for a single-socket server and does not consider the complexities associated with NUMA architectures.

Several studies have focused on dynamic resource optimization for co-running HPC applications by introducing modifications within parallel runtimes tailored to specific parallel programming models [6], [18], [4]. These approaches typically leverage hardware Performance Monitoring Counters (PMCs) to support dynamic core allocation and thread placement. AMCilk [18] and nOS-V [4] proposed system-wide task scheduling runtimes specifically for co-running applications that adopt task-based concurrency decomposition techniques. AMCilk is a modification of the Cilk parallel programming model [29], whereas nOS-V is designed to be agnostic to any specific task-based programming model. A key limitation of both is their incompatibility with thread-based parallelism, such as OpenMP work-sharing constructs. In contrast, NuPoCo [6] and LIRA [30] integrate a cooperative scheduler within the GNU OpenMP runtime to support dynamic thread placement for co-running applications. NuPoCo additionally optimizes application-level core allocation. MAPPER [2] targets similar optimization goals as NuPoCo but relies on one-time machine characterization using microbenchmarks and is independent of the underlying parallel programming model. TPLE [17] employs a learning-based model to predict optimal thread placement for a single application running on NUMA systems.

Most HPC research aimed at improving energy efficiency

TABLE II
APPLICATIONS USED FOR THE EVALUATION OF HARMONIZER

Benchmarks	Description	Configuration	Benchmark Characteristic
CoHMM (HCLib)	Heterogeneous Multiscale Method	input2.txt, HMM time steps 5000, MD time steps 20000	Neutral (N)
CoMD (OpenMP)	Classical molecular dynamics algorithms and workloads	-x 16 -y 16 -z 16 -N 200000 -D 10	Neutral (N)
SimpleMOC (OpenMP)	3D neutron transport calculations	default.in, n_egroups=10, decomp_assemblies_ax=5, assembly_width=2 x 15 cm, seg_per_track=48, cai=27	Cache Sensitive (C)
MinTally (OpenMP)	Monte Carlo neutron transport Mini Application	-p 1000000000 -f 1000 -s 1 -n 1 -e 4	Cache Sensitive (C)
XSBench (OpenMP)	Monte Carlo neutronics Mini-application	-G nuclide -g 800 -l 900 -p 700000	Cache Sensitive (C)
HPCCG (OpenMP)	HPC Conjugate Gradients	nx=256, ny=256, nz=256, max_iter = 1200	Memory Bandwidth Sensitive (M)
MiniFE (Kokkos)	Finite Element Mini-Application	nx=256, ny=256, nz=512, max_iters=500	Memory Bandwidth Sensitive (M)

in multicore processors has focused on using DVFS [10], [31], UFS [19], DCT [32], or a combination of these techniques [20], [8], [9], [21], [11]. The solutions in [32], [31], [20], [21] are tailored for applications that adopt task-based concurrency decomposition. Approaches in [9], [11] employ machine learning models for energy-aware decision-making. While several of these efforts [31], [9], [10], [11] support energy-efficient execution for co-running applications, none of them optimize thread placement or jointly coordinate DCT, DVFS, and UFS for holistic resource management. While Harmonizer shares the goals of DVFS/UFS tuning with FCUFS [11], and thread placement and DCT with MAPPER [2], it offers a more comprehensive solution for resource allocation in co-running applications. It integrates thread placement, core allocation, DVFS, and UFS without relying on prior knowledge of the hardware, application characteristics, or machine learning models. Harmonizer is designed to strike a balance between performance and energy efficiency, with the goal of maximizing system throughput while minimizing energy consumption.

III. EXPERIMENTAL METHODOLOGY

Before presenting the motivating analysis for Harmonizer, we first describe our experimental setup and the benchmarks used. We selected seven exascale proxy applications from the Exascale Computing Project (ECP) [22] for evaluation, as listed in Table II. These applications were chosen based on two key criteria: support for multicore parallelism and diversity in runtime characteristics (Section IV). To control the execution times on our system, we modified their default parameters, as mentioned in Table II. We used a variety of parallel programming models (HCLib [33], Kokkos [34], and OpenMP [35]) to demonstrate that Harmonizer operates independently of both the programming model and the parallel runtime system. All applications were compiled using LLVM 17.0.0 with the `-O3 optimization flag`. None of our applications explicitly use SIMD or AVX execution units, although the `-O3` compiler optimization flag may implicitly enable their use.

Experiments were conducted on a quad socket Intel Xeon Gold 5318H processor with 72 physical cores (18 cores per socket), each supporting two hardware threads (total of 144 logical cores). The system had 512 GB RAM and ran Ubuntu 20.04.2 LTS. Turbo Boost was disabled to minimize the performance variability [8], [9], [11]. The processor supported non-turbo core frequencies from 1.0 GHz (CF_{Min}) to 2.5 GHz (CF_{Max}) and uncore frequencies from 1.1 GHz (UF_{Min}) to 2.4 GHz (UF_{Max}), both adjustable in 0.1 GHz steps from userspace. We carried out UFS tuning by dynamically reading and writing the UFS MSRs `0x621` and `0x620`, respectively. In the default application executions (hereafter referred to as Default), the processor automatically modulates core and uncore frequencies. We used Intel's RAPL tool [36] to monitor energy consumption at the processor package level (per socket) using the *MSR_PKG_ENERGY_STATUS* counter. It includes the energy used by both core and uncore components within the package. Although our evaluation uses an Intel processor, recent AMD processors also support per-core DVFS similar to Intel. However, based on publicly available documentation, it is unclear whether they provide support for UFS or uncore PMU measurements.

IV. MOTIVATING ANALYSIS

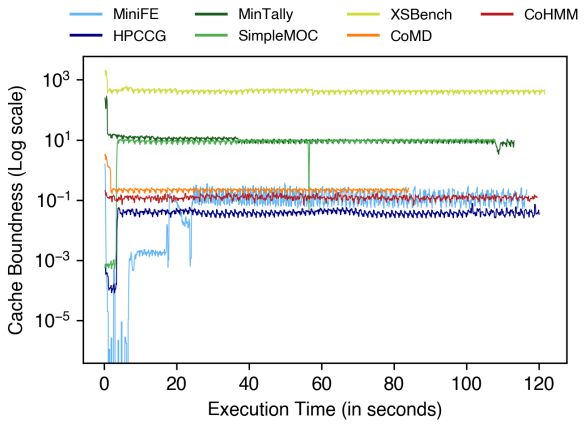
This section examines how system resource utilization varies across HPC workloads and outlines the rationale behind Harmonizer's design. We begin by analyzing the memory access patterns of our chosen applications, followed by a discussion of the impact of DVFS, UFS, and DCT on their energy consumption.

A. Analysis of LLC and memory accesses

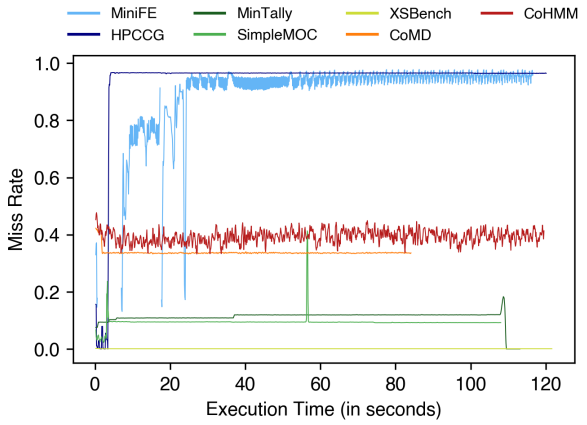
Harmonizer utilizes the Linux *perf* tool for online profiling of cache-boundness (CB) and cache miss rate (CMR) to characterize an application's memory access pattern, as defined in Equations 1 and 2, respectively. This profiling information guides its dynamic thread placement decisions. CB is defined as the ratio of total last-level cache (LLC) accesses to total memory accesses. LLC accesses are measured using core-level PMCs. Total memory accesses are measured using socket-level uncore PMCs, which include read and write requests to the integrated memory controller (IMC) and memory accesses that bypass intermediate cache levels. A higher CB value indicates that an application depends more on cache than on main memory. In contrast, a higher CMR value signifies the opposite, with greater dependence on main memory than on cache. Conversely, lower values for CB and CMR suggest the reverse trends.

$$\text{Cache Boundness (CB)} = \frac{\begin{array}{c}\texttt{MEM\_LOAD\_UOPS\_RETIRED.L3\_HIT+}\\ \texttt{MEM\_LOAD\_UOPS\_RETIRED.L3\_MISS}\end{array}}{\begin{array}{c}\texttt{UNC\_M2\_IMC\_READS.ALL+}\\ \texttt{UNC\_M2\_IMC\_WRITES.ALL+}\\ \texttt{UNC\_C\_BYPASS\_CHA\_IMC.TAKEN}\end{array}} \quad (1)$$

$$\text{Cache Miss Rate (CMR)} = \frac{\texttt{MEM\_LOAD\_UOPS\_RETIRED.L3\_MISS}}{\begin{array}{c}\texttt{MEM\_LOAD\_UOPS\_RETIRED.L3\_HIT+}\\ \texttt{MEM\_LOAD\_UOPS\_RETIRED.L3\_MISS}\end{array}} \quad (2)$$



(a) Cache boundness (CB) indicating reliance on LLC vs. DRAM



(b) Cache miss rate (CMR) indicating sensitivity to LLC

Fig. 1. Metrics to classify memory access pattern of each application

TABLE III
COMPARISON OF CMR AND CB FOR APPLICATIONS IN SOLO V/S CO-RUNNING EXECUTION

Metric	Applications Mix						Applications Mix					
	CoHMM		XSBench		SimpleMOC		CoMD		MiniFE		SimpleMOC	
	Solo	Mix	Solo	Mix	Solo	Mix	Solo	Mix	Solo	Mix	Solo	Mix
CMR	0.39	0.44	0.0009	0.003	0.09	0.36	0.34	0.73	0.9	0.97	0.09	0.17
CB	0.12	0.019	429.35	71.90	8.89	0.59	0.24	0.05	0.12	0.03	8.89	5.99

Figures 1(a) and 1(b) present the CB and CMR for isolated execution of each application. The initial noticeable downward spikes in CB (Figure 1(a)) in MiniFE occur during the load-imbalanced phase (LIP), where memory is allocated and initialized. During this phase, only a subset of cores was used. In contrast, the load-balanced phase (LBP) fully utilized all allocated cores. The other six applications, SimpleMOC, XSBench, HPCCG, MinTally, CoMD, and CoHMM, primarily exhibited LBP, with only a minor LIP at the start. We placed **MiniFE** and **HPCCG** in **memory-bound** category due to extremely high CMR (Figure 1(b)). Likewise, **XSBench**, **MinTally** and **SimpleMOC** were placed in **cache-sensitive** category due to very low CMR. **CoHMM** and **CoMD** fall into **neutral** category as their CMR is close to mid value 0.5.

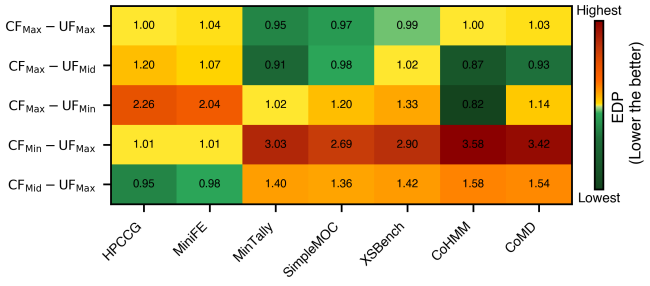


Fig. 2. EDP relative to Default at different DVFS+UFS settings

Resource constraints can influence an application's CB and CMR values, causing them to vary between isolated and co-running execution scenarios. Table III compares these metrics during the load-balanced phase (LBP) under two conditions: isolated execution (Figure 1) and co-running execution in groups of three applications, where one mix is evaluated at a time. In both cases, each application used 48 threads mapped to 48 hardware contexts, ensuring full system utilization (144 hardware contexts) across all mixes. During co-execution with XSBench, both CoHMM and SimpleMOC exhibited CMR values near 0.5, which could incorrectly suggest that SimpleMOC falls into the neutral category. However, unlike CoHMM, SimpleMOC has a CB value much closer to one, correctly identifying it as cache-sensitive. In contrast, for the other application mix, CMR alone was sufficient to accurately classify each application's memory access behaviour.

B. Impact of frequency scaling on energy efficiency

To analyze the impact of DVFS and UFS on energy savings, we executed each application in isolation using different sets of core (CF) and uncore (UF) frequencies. We then compared the Energy-Delay Product (EDP) against the default execution scenario, where the processor automatically modulates core and uncore frequencies. EDP is a widely used metric that quantifies the trade-off between performance and energy consumption [8]. It is calculated as the product of an application's total energy usage and its execution time. Lower EDP values indicate greater overall efficiency, as they represent reduced energy consumption, faster execution, or an effective balance between the two. Figure 2 summarizes the results of this experiment. We observe that the impact of adjusting core and uncore frequencies on EDP varies with the application's inherent memory access characteristics. For memory-bound applications, reducing the core frequency from maximum to minimum improves EDP. In contrast, cache-sensitive and neutral applications benefit from increasing the core frequency from minimum to maximum. XSBench favours CF_{Max} and UF_{Max} due to its highly cache-bound behaviour (uncore part).

C. Impact of core allocation on energy efficiency

Confining a fixed number of application threads to a variable number of active cores is a widely adopted strategy for Dynamic Concurrency Throttling (DCT) [7]. Although this

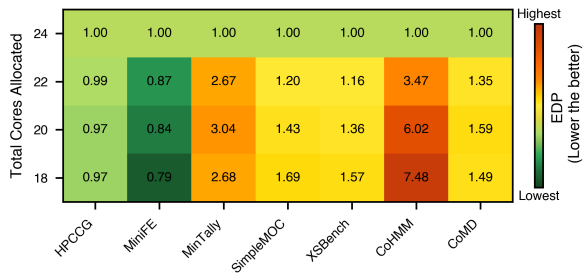


Fig. 3. EDP relative to Default at varying number of cores

approach oversubscribes threads on cores, it avoids modifying the parallel runtime to adjust the active thread count dynamically. DCT is particularly beneficial for memory-bound applications, where memory bandwidth becomes the performance bottleneck. To evaluate the impact of DCT on our selected applications, we co-executed each one with two instances of the STREAM benchmark, forming a mix of three co-runners. We chose STREAM to impose pressure on the memory subsystem [20]. Threads of each co-runner were evenly distributed across the four sockets in our system (48 threads per application, mapped to 48 hardware contexts). Figure 3 presents the results of this experiment, showing the effect of varying core allocations for our application while keeping STREAM's allocation fixed. We observe that reducing core allocation improves EDP only for the memory-bound applications (HPCCG and MiniFE). For all other applications, even a slight reduction in cores results in a degradation of EDP.

V. Design and Implementation

The previous section demonstrated that memory access patterns and system resource utilization vary across HPC applications. Our implementation utilizes a lightweight daemon, Harmoniser, which is launched alongside the parallel applications. It periodically monitors application-level metrics CB, CMR, and IPS (Instructions retired per Second), as well as socket-level energy consumption via PMCs. Harmonizer activates at fixed intervals (every 100ms) to limit interference with application execution and invokes its dynamic resource optimization policy (Figure 4) during each cycle. No modifications to applications are required to use Harmonizer. We illustrate this policy using an example with three co-running applications on a four-socket system ($Socket_A$, $Socket_B$, $Socket_C$, $Socket_D$), each with six cores and two hardware contexts per core. Harmonizer distributes CPUs equally among all threads in a round-robin manner (**TP_Policy$_{RR}$**, Figure 4(a)). Each application spawns 16 threads, with two mapped to the hardware contexts of the same physical core. Green threads represent a memory-bound application App_A (e.g., MiniFE), yellow indicates a neutral application App_B (e.g., CoMD), and red represents a cache-sensitive application App_C (e.g., SimpleMOC). Initially, the core and uncore frequencies of each socket are set to their maximum values. When App_A enters LIP (Figure 4(b)), Harmonizer detects reduced CPU utilization and reallocates App_A's idle cores to co-runners in their LBP on the same socket. It is done via thread unpacking, migrating one thread each from App_B and App_C. When App_A transitions back to LBP, its threads resume CPU usage, leading to temporary oversubscription (Figure 4(c)). Harmonizer detects this shift through a rise in IPS on the affected CPUs and restores the original mapping as shown in Figure 4(a).

A. Thread placement policy (TP_Policy)

Once all applications enter LBP, the inactive threads of an application get activated, increasing the IPS at the previously idle cores (Figure 4(c)). Harmonizer detects this transition to LBP and initiates the exploration for each application's optimal thread placement policy (TP_Policy). It requires accurately profiling CB and CMR by minimizing LLC sharing. To achieve this, Harmonizer uses a refined thread placement strategy, **TP_Policy$_{Sock}$** (Figure 4(d)). The TP_Policy$_{Sock}$ ensures a 1-to-1 mapping of application threads to hardware contexts within a socket. If threads of an application exceed the available hardware contexts on one socket, the surplus is pinned to the last socket ($Socket_D$). Additionally, TP_Policy$_{Sock}$ facilitates CB profiling by leveraging uncore PMCs (socket-level events). Any mixes having TP_Policy$_{Sock}$ as the optimal thread placement, an additional benefit from TP_Policy$_{Sock}$ is reduced energy usage by setting application-specific UFS at each socket (see Section V-B).

Algorithm 1: Determining application type based on its CMR and CB

```
Function: ApplicationType(CMR, CB)
ReferencePoints_CMR ← {0, 0.5, 1};
Distances_CMR ← ComputeEuclideanDistances(CMR, ReferencePoints_CMR);
NearestRefPt_CMR ← NearestNeighbor(Distances_CMR, ReferencePoints_CMR);
if NearestRefPt_CMR = 1 then
    return MEMORY;
else
    FirstCheck ← None, SecondCheck ← None;
    if NearestRefPt_CMR = 0 then
        FirstCheck ← CACHE;
    end
    ReferencePoints_CB ← {0, 1};
    Distances_CB ← ComputeEuclideanDistances(CB, ReferencePoints_CB);
    NearestRefPt_CB ← NearestNeighbor(Distances_CB, ReferencePoints_CB);
    if NearestRefPt_CB = 0 then
        SecondCheck ← NEUTRAL;
    else
        SecondCheck ← CACHE;
    end
    if FirstCheck = CACHE and SecondCheck = CACHE then
        return CACHE;
    else
        return SecondCheck;
    end
end
```

The variability in CMR and CB values across co-running applications complicates direct comparisons of their memory access patterns. Hence, the Harmonizer utilizes Algorithm 1 to dynamically classify applications based on online profiling of CMR and CB values. The Algorithm 1 uses two levels of nearest neighbour clustering [37] to classify the applications into cache-bound, neutral and memory-bound categories. An application's CMR is used at the first level to calculate its

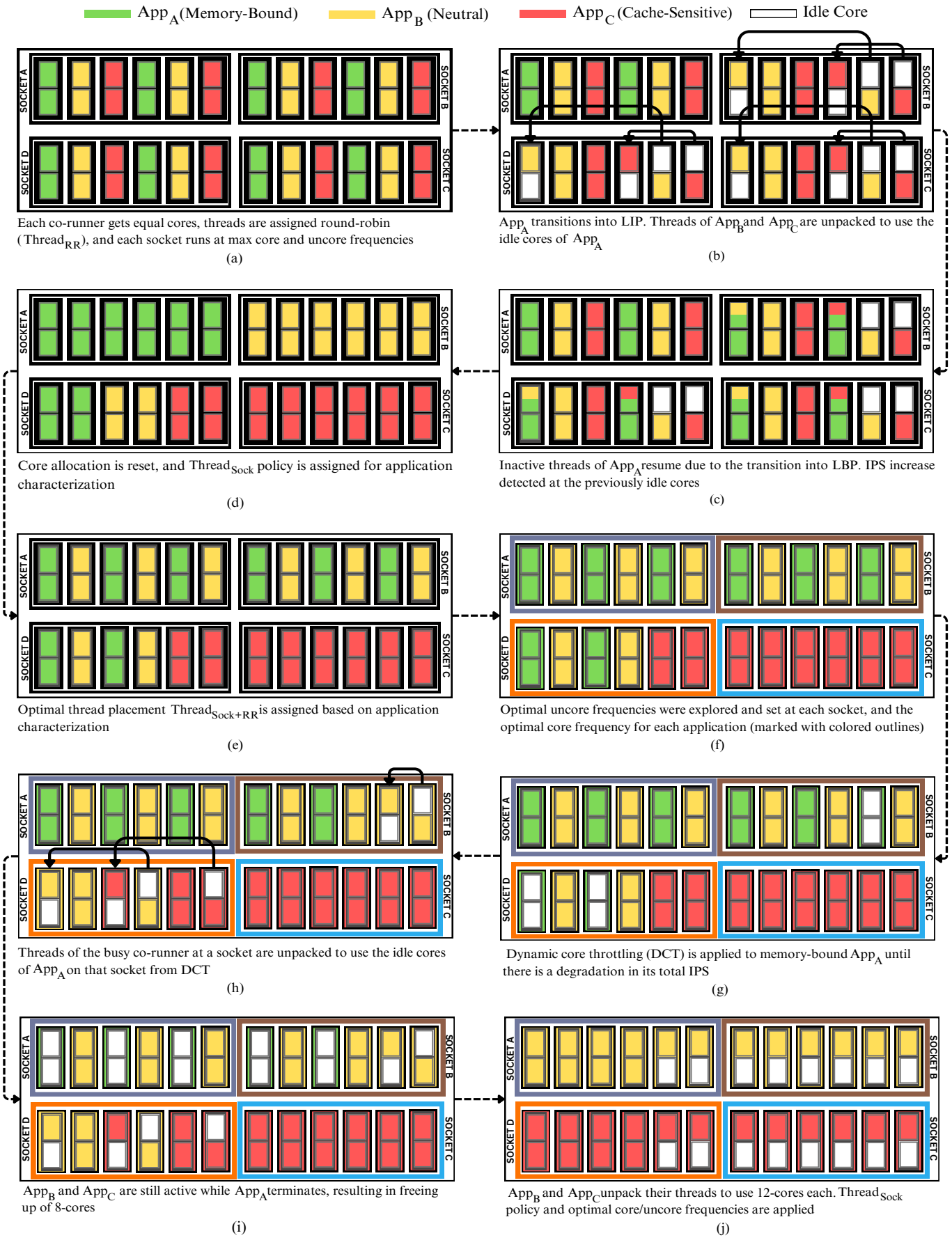


Fig. 4. Harmonizer policies for three co-running applications on a quad-socket system

Euclidian distances from the predefined reference points 0, 0.5 and 1, each representing low, moderate and high miss rates, respectively. The resultant nearest reference point determines the application type. As discussed in Section IV, relying solely on CMR alone can lead to misclassification when it is close to 0.5. Hence, the application's CB is used at the second level of nearest neighbour clustering to calculate its Euclidian distances from the predefined reference points 0 (neutral) and 1 (cache-sensitive).

We observed that when all threads of a co-runner are confined to a single socket (e.g., in a four-application mix with each co-runner mapped to a single socket), the CMR of the neutral application tends to zero, which can lead to the misclassification of a cache-sensitive application. To mitigate this, we implement a two-level check to differentiate between neutral and cache-sensitive types (Line 8, Algorithm 1). The first check evaluates the application's CMR (Line 9). If it is close to zero, the application is provisionally marked as cache-sensitive (Line 10). The second check considers the application's CB. If its Euclidean distance is close to zero, the application is reclassified as neutral. Otherwise, it remains cache-sensitive. Thus, an application is finally classified as cache-sensitive only if both checks indicate it as such (Line 20).

To mitigate this, we implement a two-level check to differentiate between neutral and cache-sensitive types (Line 8, Algorithm 1). The first check evaluates the application's CMR (Line 9). If it is close to zero, the application is provisionally marked as cache-sensitive (Line 10). The second check considers the application's CB. If its Euclidean distance is close to zero, the application is reclassified as neutral. Otherwise, it remains cache-sensitive. Thus, an application is finally classified as cache-sensitive only if both checks indicate it as such (Line 20).

The Harmonizer uses Algorithm 1 to classify App_A, App_B, and App_C as memory-bound, neutral, and cache-sensitive applications. The thread placement policy for a cache-sensitive application involves contiguous allocation on a single socket to minimize cache misses, with extension to another socket only when the allocated cores exceed the core count of the original socket. For memory-bound applications, threads are distributed in a round-robin manner across available sockets to improve memory bandwidth, depending on socket availability after prioritizing cache-sensitive applications (if any). Threads of neutral application can be allocated in any order on any socket. For the mix shown in Figure 4, App_C is assigned a contiguous placement. At the same time, the threads of the other two applications are distributed in a round-robin fashion over the three sockets. This thread placement policy is referred to as $TP\_Policy_{Sock+RR}$ (see Figure 4(e)). If all three applications were cache-sensitive, Harmonizer would retain the TP_Policy_{Sock} policy (Figure 4(d)) to enable uncore frequency scaling (Section V-B). If all applications were memory-bound, Harmonizer would select TP_Policy_{RR} (Figure 4(a)). During the course of application execution, Harmonizer will attempt to reclassify an application type only if its CB and CMR change by more than 10%.

B. Processor frequency scaling policy (CF_Policy and UF_Policy)

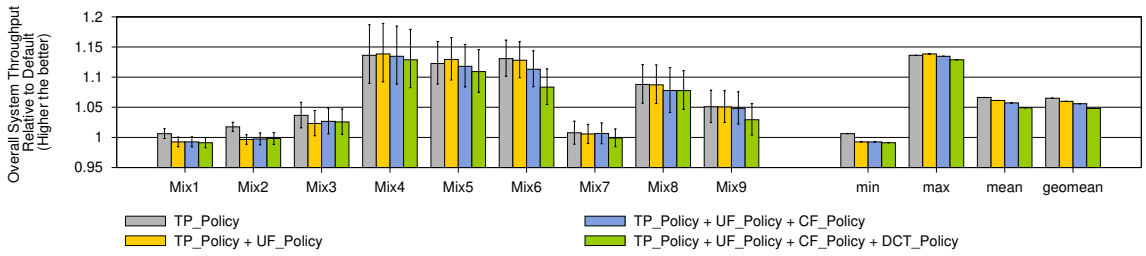
After optimizing thread placement, Harmonizer reduces the energy consumption by setting the optimal uncore frequency using UFS for each socket (UF_Policy), followed by core frequencies for each co-runner (CF_Policy) using DVFS, as shown in Figure 4(f). As shown in Figure 2, it leverages application characterization (Algorithm 1) to minimize exploration steps for frequency tuning. For memory-bound applications, it starts UFS from the UF_{Max} and core frequency from the CF_{Mid}. For cache-sensitive applications, UFS begins from UF_{Mid} and DVFS from the CF_{Max}. In neutral applications, UFS starts from UF_{Min}, while core frequency begins from CF_{Max}. Frequency exploration continues iteratively until a degradation in IPS is detected. The threshold for tolerating degradation was 1% for UFS and 0.1% for DVFS, as the latter is highly sensitive to performance. When degradation is observed, the previously set frequency is set as optimal. Since UFS operates at the socket level, UF_{Opt} is explored only for TP_Policy_{Sock} and $TP\_Policy_{Sock+RR}$ thread placements and only on sockets exclusively hosting a single application. Sockets shared by multiple applications always run at UF_{Max}. Once the UF_{Opt} is determined, DVFS exploration is performed for each application, regardless of the TP_Policy.

C. Dynamic concurrency throttling policy (DCT_Policy)

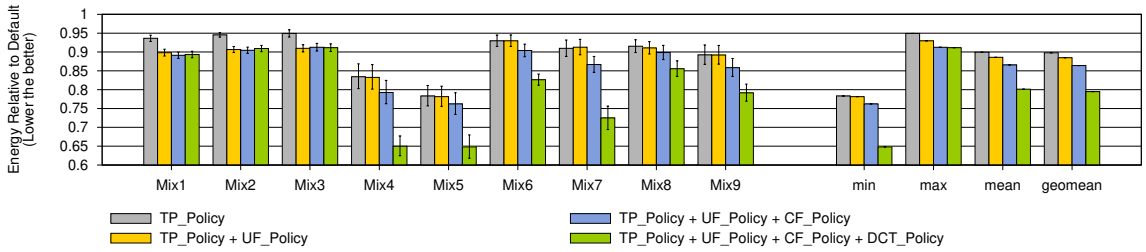
After setting UF_{Opt} and CF_{Opt}, Harmonizer initiates dynamic concurrency throttling (DCT_Policy) for all memory-bound co-runners (only), such as App_A in our example (Figure 4(g)). Memory-bound applications typically exhibit lower per-core IPS than compute-bound ones due to frequent CPU stalls from memory access latency (see Section V-C). Hence, Harmonizer attempts DCT only in memory-bound applications to enhance per-core utilization and reduce memory bandwidth contention. Using thread-packing, it gradually reduces App_A's core allocation, decreasing by one core per step until the degradation in IPS exceeds 1% from the original allocation. Each released core is reassigned to a co-running application on the same socket, after which Harmonizer configures the CF_{Opt} for the migrated threads (Figure 4(h)). Harmonizer continuously monitors the IPS of App_A to ensure performance stability. If IPS degradation is detected, the previous thread mapping and core frequency settings are restored. When a co-runner terminates, Harmonizer dynamically reconfigures core allocations for the remaining applications and restarts the resource optimization. As shown in Figure 4(i), App_A has terminated, so the core allocations for App_B and App_C increase from eight to twelve through thread unpacking. Harmonizer then recalculates the thread placement policy and adopts TP_Policy_{Sock}, followed by setting the UF_{Opt} and CF_{Opt} for each application (Figure 4(j)). This process continues until all applications have completed execution, at which point Harmonizer terminates its operation.

TABLE IV
CO-RUNNING APPLICATION MIXES AND TYPES: CACHE(C), MEMORY(M), NEUTRAL(N)

Name (Memory access pattern)	Mix1 (NCC)	Mix2 (NCCC)	Mix3 (NCN)	Mix4 (MNM)	Mix5 (MMNN)	Mix6 (NMN)	Mix7 (CMM)	Mix8 (MCNC)	Mix9 (CMN)
App_A	CoHMM	CoHMM	CoMD	HPCCG	HPCCG	CoMD	XSBench	MiniFE	SimpleMOC
App_B	XSBench	MinTally	XSBench	CoHMM	MiniFE	HPCCG	MiniFE	XSBench	MiniFE
App_C	SimpleMOC	XSBench	CoHMM	MiniFE	CoHMM	CoHMM	HPCCG	CoMD	CoMD
App_D	—	SimpleMOC	—	—	CoMD	—	—	MinTally	—
Default Time (s)	128.7	163.6	125.1	153	172.6	126.8	153	162.9	128



(a) Improvement in system throughput with no significant degradation



(b) Reduction in in total processor package energy consumption

Fig. 5. Experimental evaluation for each Harmonizer policy

VI. EXPERIMENTAL EVALUATION

This section presents the experimental evaluation of Harmonizer using the co-running mixes listed in Table IV. We created mixes to ensure diversity in memory access patterns, an equal distribution across the three TP_Policy categories, and representation of both three- and four-application co-runners per category. Each co-running application in each mix completes its execution nearly simultaneously. We first analyze the impact of each Harmonizer policy on system throughput and energy consumption relative to the Default execution. We then compare Harmonizer against two recent solutions, NuPoCo [6] and MAPPER [2]. The artifact of this paper is available online [38].

A. Evaluation of Harmonizer policies

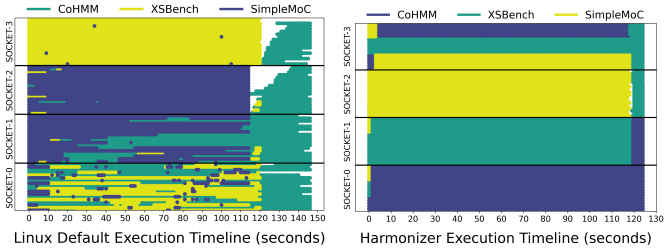
$$\text{Speedup}_{\text{Mix}} = \frac{\left(\prod_{i=0}^{n-1} \text{Time}_{\text{Default}_{\text{App}_i}}\right)^{\frac{1}{n}}}{\left(\prod_{i=0}^{n-1} \text{Time}_{\text{Harmonizer}_{\text{App}_i}}\right)^{\frac{1}{n}}} \quad (3)$$

Figure 5 illustrates the impact of Harmonizer's policies on overall system throughput and energy consumption. The improvement in throughput for each co-running mix, relative to its default execution, was computed using the geometric mean formula presented in Equation 3 [16]. The reported energy usage is the sum of energy used across all four sockets, measured from the start of each mix's execution until all co-running applications are complete. Table V presents the Core Frequency (CF) and Uncore Frequency (UF) set by Harmonizer (with all four policies), along with the reduction in core allocation using DCT and its duration for each application in each mix. Figure 6 illustrates the Default execution of one representative mix from each of the three thread placement policies selected by Harmonizer (discussed below) and compares them with the corresponding Harmonizer-managed execution. We used the Linux *perf* tool's *sched:sched_switch* tracepoint to capture context switch events of application threads in this visualization. Since context switches are rare in Harmonizer, the recorded data points were sparse. To address this, we applied forward data propagation, extending the previous timestamp until a new data point was available, ensuring a continuous representation of thread placements.

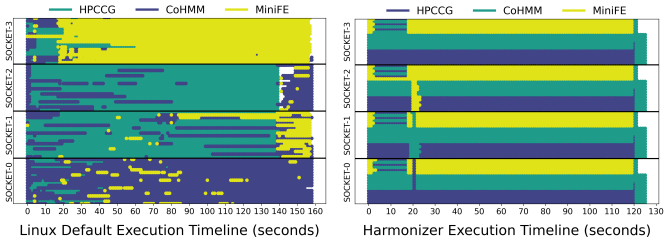
Harmonizer configured $\text{TP\_Policy}_{\text{Sock}}$ for Mix1–Mix3 (Group-1), $\text{TP\_Policy}_{\text{RR}}$ for Mix4–Mix6 (Group-2), and $\text{TP\_Policy}_{\text{Sock+RR}}$ for Mix7–Mix9 (Group-3). We can observe from Figure 6 that the profiling phase in Harmonizer was negligible, enabling the quick calculation of the optimal thread placement, unlike Default, where thread interferences persist over longer execution durations. The throughput gains were minimal in Group-1 (0.6%–3.7%) as the Default thread

TABLE V
PER-APPLICATION DVFS, UFS, AND DCT SETTINGS IN EACH MIX WITH HARMONIZER

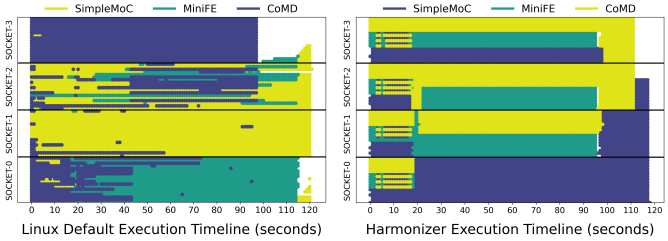
Application	Metric	Mix1	Mix2	Mix3	Mix4	Mix5	Mix6	Mix7	Mix8	Mix9
App_A	CF (GHz)	Max	Max	Max	2.3	2.2	Max	Max	2.2	Max
	UF (GHz)	1.1	1.1	2	Max	Max	Max	Max	Max	Max
	Core reduction	0	0	0	2 (22.5%)	2 (15.9%)	0	0	2 (8.1%)	0
App_B	CF (GHz)	Max	Max	Max	Max	2.0	1.9	2.0	Max	1.9
	UF (GHz)	2.3	2.3	Max	Max	Max	Max	Max	2.3	Max
	Core reduction	0	0	0	0	2 (8.4%)	2 (14.2%)	2 (12.2%)	0	3 (13.6%)
App_C	CF (GHz)	Max	Max	Max	1.9	2.4	Max	1.9	Max	Max
	UF (GHz)	2.3	Max	1.4	Max	Max	Max	Max	Max	Max
	Core reduction	0	0	0	3 (17.1%)	0	0	2 (16.5%)	0	0
App_D	CF (GHz)	-	Max	-	-	Max	-	-	Max	-
	UF (GHz)	Max	2.3	Max	Max	Max	Max	Max	Max	Max
	Core reduction	-	-	-	-	0	-	-	0	-



(a) Mix1



(b) Mix4



(c) Mix9

Fig. 6. Default vs. Harmonizer thread placement

placement was similar to that of Harmonizer (TP_Policy_{Sock}), albeit with several thread migrations (see Figure 6(a)). Cache-sensitive applications dominate Mix1 and Mix2, resulting in contention for the LLC, whereas the neutral application dominance in Mix3 enables it to perform better than the other two. In the other two groups, thread placement configured by Harmonizer was organized according to the benchmark categories (unlike the Default, see Figure 6(b) and Figure 6(c)), leading to higher improvements in throughput (12%–13.6%) in Group-2 and (0.8%–8.8%) in Group-3. Energy savings from TP_Policy (Figure 8(b)) correlated with throughput gains. Hence, it was the highest in Group-2 (21.7%), followed by Group-3 (10.8%) and Group-1 (6.4%).

We can observe that the throughput improvement with just the TP_Policy is better than when using all four policies. Recall from Section V-B that Harmonizer uses IPS for explorations inside UF_Policy, CF_Policy, and DCT_Policy, and the threshold to tolerate changes is stricter for DVFS than UFS and DCT. In Group-1, where no memory-bound applications exist, UF_Policy successfully lowered uncore frequencies for nearly every application (see Table V), leading to a slight throughput degradation (1.4%–2.1%) compared to using TP_Policy alone. In Group-2, Harmonizer did not initiate UF_Policy, as each socket hosted threads from multiple applications due to TP_Policy_{RR}, resulting in similar throughput using only TP_Policy or both TP_Policy and UF_Policy. The same reasoning applies to Group-3. In Group-3, UF_Policy set UF_{Max} almost everywhere, even for sockets running cache-sensitive applications, due to the increased sensitivity of IPS to uncore frequency for the memory access pattern in these mixes. The energy savings from UF_Policy depend on both the number of sockets where it can lower the uncore frequency and the extent of that reduction (see Table V). Consequently, UF_Policy achieves energy reduction only in Group 1, with savings of up to 4% compared to TP_Policy.

Recall from Section IV-B that lowering core frequency in neutral and cache-sensitive applications leads to performance degradation. A similar effect in IPS was observed for these benchmarks with CF_Policy in all mixes. However, unlike UF_Policy, CF_Policy did not result in noticeable performance degradation due to a stricter DVFS exploration threshold. As a result, CF_Policy reduced core frequencies only in mixes containing memory-bound applications (Group-2 and Group-3). Additional energy savings with CF_Policy alone ranged between 1.4% (Mix8) to 5.0% (Mix7). DCT_Policy, like CF_Policy, applies only to mixes with memory-bound applications (Group-2 and Group-3) but uses a moderate threshold for optimal core count exploration. However, similar to UF_Policy, DCT_Policy also leads to a slight degradation in throughput. We observed it to be 2.7% in Mix6, 1.8% in Mix9, and below 0.8% in Mix4, Mix5 and Mix7. In Mix6 and Mix9, the CoMD benchmark exhibited characteristics of a light memory-bound application, due to which DCT_Policy attempted concurrency throttling but quickly revoked its decision due to degradation (see Table V). Additional energy

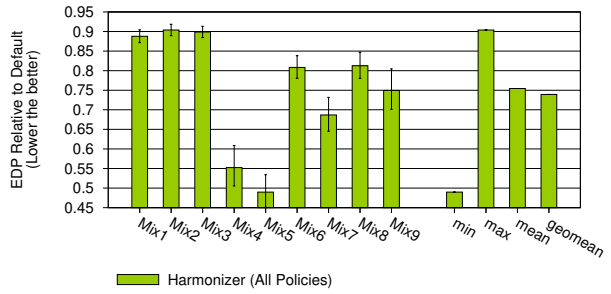


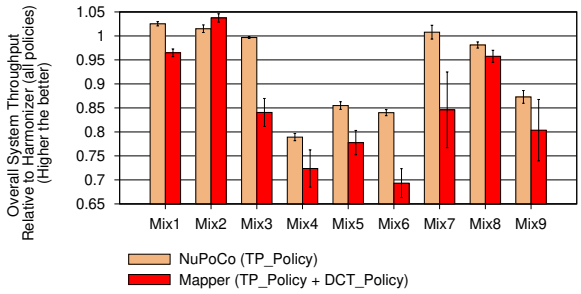
Fig. 7. Harmonizer EDP (Turnaround time×Energy) relative to Default

savings with DCT_Policy ranged between 4.8% (Mix8) and 18% (Mix4). Energy savings from DCT_Policy were less in Mix6 than in Mix4 and Mix5 due to the presence of a single memory-bound application in Mix6, as compared to two in the other two mixes.

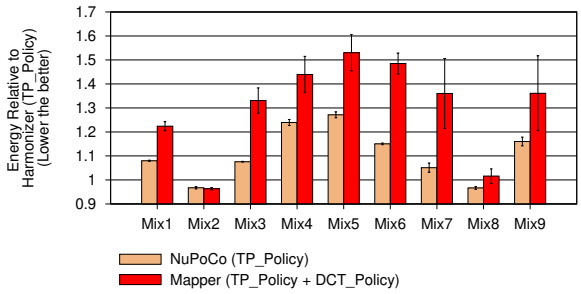
The primary goal of TP_Policy is to minimize the execution time by reducing contention at shared resources. However, it will also result in a reduction in energy consumption. In contrast, the primary goal of CF_Policy, UF_Policy, and DCT_Policy is to reduce the energy consumption while balancing the performance of the mix. We can observe from Figure 8(b) that the geometric mean energy savings from TP_Policy relative to Default were 10%. For Group-1 mixes, UF_Policy achieved up to 4% extra energy savings relative to TP_Policy. CF_Policy and DCT_Policy achieved geomean energy savings of 3.8% and 11.5%, respectively, over TP_Policy. To demonstrate the combined effect of reduction on execution time and energy consumption, Figure 7 presents the EDP obtained by Harmonizer relative to the Default execution. The EDP is calculated as the product of turnaround time and total energy consumption of each mix. We can observe that geometric mean savings in EDP by Harmonizer were 26%. We also measured the profiling overhead in Harmonizer. Running a mix with and without the daemon profiler resulted in a 1.7% overhead (geometric mean across all mixes). The total time required to determine thread placement in each mix was 0.5s and was performed only once, since the CM/CMR values did not vary beyond the 10% threshold (see Section V-A). Likewise, the total exploration time for DVFS and UFS across all mixes ranged from 1.0s (Mix1) to 2.8s (Mix5) for DVFS, and from 0s (Mix4, Mix5, Mix6) to 3.2s (Mix2) for UFS. Harmonizer can support more than four co-running applications. However, the UF_Policy may not benefit when multiple applications with varying memory access patterns are placed over the same socket.

B. Comparison with NuPoCo and MAPPER

Figure 8 compares the overall system throughput and energy consumption of NuPoCo and MAPPER relative to Harmonizer. NuPoCo was integrated into the OpenMP runtime, so we could not utilize its DCT_Policy. Instead, we implemented a standalone TP_Policy as described in the paper [6]. We used the open-source implementation of Mapper [2], which includes its standalone implementations of TP_Policy and DCT_Policy.



(a) Harmonizer achieved similar or better throughput



(b) Harmonizer achieved significantly better energy savings

Fig. 8. Comparison of Harmonizer with state-of-the-art implementations

We compared throughput using Harmonizer with all four policies enabled to ensure a fair evaluation, as only TP_Policy in Harmonizer enhances system throughput. Similarly, for comparing energy consumption, we used Harmonizer with TP_Policy only for fairness. NuPoCo and MAPPER optimize thread placement with the goal of minimizing the CMR at each socket, iteratively adjusting placements until CMR stabilizes. In Group-2, which consists of mixes of neutral and memory-bound benchmarks (i.e., workloads with inherently low and high CMR), this disparity causes NuPoCo and MAPPER to perform excessive thread migrations, leading to significant throughput degradation. In contrast, Group-1 features cache-sensitive and neutral benchmarks, enabling CMR to stabilize more quickly and resulting in performance comparable to Harmonizer (except for MAPPER in Mix3). In Group-3, Mix9 includes all three benchmark categories, resulting in frequent thread migrations and the highest throughput degradation among this group. While Mix8 also encompasses all three categories, limited thread movement between the memory-bound MiniFE and the neutral CoMD allows both NuPoCo and MAPPER to perform similarly to Harmonizer. We did not integrate our CF_Policy and UF_Policy with NuPoCo or MAPPER for comparison, as their designs inherently cause frequent thread migrations. For instance, in our mixes, NuPoCo performed an average of 157 thread migrations to validate optimal placement.

VII. Conclusion

Efficient resource management in modern multiprocessor systems is crucial for maximizing throughput and minimizing energy consumption. This paper presents Harmonizer, a programming-model-agnostic tool for co-running applications

on multicore and multi-socket servers that dynamically optimizes thread placement, core/uncore frequencies, and core allocation without requiring prior knowledge of application workloads or memory access patterns. Our results show that Harmonizer offers a one-stop solution for improving throughput and reducing energy usage compared to traditional approaches. As future work, we plan to extend Harmonizer to support applications with dynamically varying memory access patterns and scale its use to cluster-level environments.

VIII. Acknowledgments

This research was funded by a research grant from Shell India Markets Private Ltd., CW764055. The authors are grateful to the anonymous reviewers for their suggestions on improving the paper's presentation. ChatGPT was used for improving grammar and spelling and for minor language polishing of the author's original text in this paper.

References

[1] "TOP500," June 2025. [Online]. Available: https://top500.org/lists/top500/2025/06/

[2] S. Srikanthan, S. Chakraborti, P. Ferro, and S. Dwarkadas, "MAPPER: Managing application performance via parallel efficiency regulation," *ACM TACO*, 2022. [Online]. Available: https://doi.org/10.1145/3501767

[3] R. Chen, H. Shi, Y. Li, X. Liu, and G. Wang, "OLPart: Online learning based resource partitioning for colocating multiple latency-critical jobs on commodity computers," in *EuroSys*, 2023. [Online]. Available: https://doi.org/10.1145/3552326.3567490

[4] D. Álvarez, K. Sala, and V. Beltran, "nOS-V: Co-executing hpc applications using system-wide task scheduling," in *IPDPS*, 2024. [Online]. Available: https://doi.org/10.1109/IPDPS57955.2024.00035

[5] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of scheduling techniques for addressing shared resources in multicore processors," *ACM Comput. Surv.*, 2012. [Online]. Available: https://doi.org/10.1145/2379776.2379780

[6] Y. Cho, C. A. C. Guzman, and B. Egger, "Maximizing system utilization via parallelism management for co-located parallel applications," in *PACT*, 2018. [Online]. Available: https://doi.org/10.1145/3243176.3243199

[7] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda, "Pack & Cap: adaptive dvfs and thread packing under power caps," in *MICRO*, 2011. [Online]. Available: https://doi.org/10.1145/2155620.2155641

[8] S. Kumar, A. Gupta, V. Kumar, and S. Bhalachandra, "Cuttlefish: library for achieving energy efficiency in multicore parallel programs," in *SC*, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476163

[9] S. Abera, M. Balakrishnan, and A. Kumar, "Performance-energy trade-off in modern cmps," *ACM TACO*, 2020. [Online]. Available: https://doi.org/10.1145/3427092

[10] T. Babakol, A. Canino, and Y. D. Liu, "Eflect: porting energy-aware applications to shared environments," in *ICSE*, 2022. [Online]. Available: https://doi.org/10.1145/3510003.3510145

[11] H. Zhang, A. Nukada, and Q. Liao, "FCUFS: Core-level frequency tuning for energy optimization on intel processors," in *CLUSTER*, 2024. [Online]. Available: https://doi.org/10.1109/CLUSTER59578.2024.00026

[12] S. Kundan, T. Marinakis, I. Anagnostopoulos, and D. Kagaris, "A pressure-aware policy for contention minimization on multicore systems," *ACM TACO*, 2022. [Online]. Available: https://doi.org/10.1145/3524616

[13] V. S. da Silva *et al.*, "Smart resource allocation of concurrent execution of parallel applications," *Wiley CCPE*, 2023. [Online]. Available: https://doi.org/10.1002/cpe.6600

[14] C. Bilbao, J. C. Saez, and M. Prieto-Matias, "Divide&Content: A fair os-level resource manager for contention balancing on numa multicores," *IEEE TPDS*, 2023. [Online]. Available: https://doi.org/10.1109/TPDS.2023.3309999

[15] T. Marinakis and I. Anagnostopoulos, "Performance and fairness improvement on cmps considering bandwidth and cache utilization," *IEEE Computer Architecture Letters*, 2019.

[16] R. B. Roy, T. Patel, and D. Tiwari, "SATORI: Efficient and fair resource partitioning by sacrificing short-term benefits for long-term gains," in *ISCA*, 2021. [Online]. Available: https://doi.org/10.1109/ISCA52012.2021.00031

[17] K. Antoniadis, R. Guerraoui, and V. Trigonakis, "Thread-Placement Learning," in *ICDCS*, 2020.

[18] Z. Wang, C. Xu, K. Agrawal, and J. Li, "AMCilk: A framework for multiprogrammed parallel workloads," in *HiPC*, 2020. [Online]. Available: https://doi.org/10.1109/HiPC50609.2020.00035

[19] N. Gholkar, F. Mueller, and B. Rountree, "Uncore power scavenger: A runtime for uncore power conservation on hpc systems," in *SC*, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356150

[20] A. Navarro Muñoz, A. F. Lorenzon, E. Ayguadé Parra, and V. Beltran Querol, "Combining dynamic concurrency throttling with voltage and frequency scaling on task-based programming models," in *ICPP*, 2021. [Online]. Available: https://doi.org/10.1145/3472456.3472471

[21] J. Chen, M. Manivannan, B. Goel, and M. Pericàs, "JOSS: Joint exploration of cpu-memory dvfs and task scheduling for energy efficiency," in *ICPP*, 2023. [Online]. Available: https://doi.org/10.1145/3605573.3605586

[22] "ECP ProxyApps," Released. [Online]. Available: https://proxyapps.exascaleproject.org/app/

[23] Intel Corporation, "Deterministic network functions virtualization with intel® resource director technology," Tech. Rep., 2016. [Online]. Available: https://builders.intel.com/docs/networkbuilders/deterministic_network_functions_virtualization_with_Intel_Resource_Director_Technology.pdf

[24] K. Nikas, N. Papadopoulou, D. Giantsidi, V. Karakostas, G. Goumas, and N. Koziris, "DICER: Diligent cache partitioning for efficient workload consolidation," in *ICPP*, 2019. [Online]. Available: https://doi.org/10.1145/3337821.3337891

[25] T. Patel and D. Tiwari, "CLITE: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers," in *HPCA*, 2020. [Online]. Available: https://doi.org/10.1109/HPCA47549.2020.00025

[26] R. Chen, J. Wu, H. Shi, Y. Li, X. Liu, and G. Wang, "DRLPart: A deep reinforcement learning framework for optimally efficient and robust resource partitioning on commodity servers," in *HPDC*, 2021. [Online]. Available: https://doi.org/10.1145/3431379.3460648

[27] R. Chen, H. Shi, J. Wu, Y. Li, X. Liu, and G. Wang, "Jointly optimizing job assignment and resource partitioning for improving system throughput in cloud datacenters," *ACM TACO*, 2023. [Online]. Available: https://doi.org/10.1145/3593055

[28] S. Chen, C. Delimitrou, and J. F. Martínez, "PARTIES: Qos-aware resource partitioning for multiple interactive services," in *ASPLOS*, 2019. [Online]. Available: https://doi.org/10.1145/3297858.3304005

[29] M. Frigo, "Multithreaded programming in Cilk," in *PASCO*, 2007.

[30] A. Collins, T. Harris, M. Cole, and C. Fensch, "LIRA: Adaptive contention-aware thread placement for parallel runtime systems," in *ROSS*, 2015. [Online]. Available: https://doi.org/10.1145/2768405.2768407

[31] H. Ribic and Y. D. Liu, "AEQUITAS: Coordinated energy management across parallel applications," in *ICS*, 2016. [Online]. Available: https://doi.org/10.1145/2925426.2926260

[32] S. Sridharan, G. Gupta, and G. S. Sohi, "Adaptive, efficient, parallel execution of parallel programs," in *PLDI*, 2014. [Online]. Available: https://doi.org/10.1145/2594291.2594292

[33] "HCLib," Accessed 2025. [Online]. Available: https://habanero-rice.github.io/hclib/

[34] "Kokkos," Accessed 2025. [Online]. Available: https://kokkos.org/

[35] "OpenMP," Accessed 2025. [Online]. Available: https://www.openmp.org/specifications/

[36] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "RAPL: memory power estimation and capping," in *ISLPED*, 2010. [Online]. Available: https://doi.org/10.1145/1840845.1840883

[37] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, 1967. [Online]. Available: https://doi.org/10.1109/TIT.1967.1053964

[38] "Energy-aware runtime resource harmonizer for co-running applications." [Online]. Available: https://doi.org/10.5281/zenodo.17481166