Harmony in Many-Cores: Synchronization-Aware QoS Control for S-NUCA Systems

Sudam M. Wasala*, Anuj Pathania*, Clemens Grelck*†, and Andy D. Pimentel*
*Informatics Institute, University of Amsterdam, The Netherlands
†Institute for Informatics, Friedrich Schiller University, Jena, Germany
Email: {s.m.wasala, a.pathania, a.d.pimentel}@uva.nl, clemens.grelck@uni-jena.de

***Abstract*—QoS management on S-NUCA many-core processors is challenging due to the physically distributed last-level caches, which introduce core-dependent access latencies and performance heterogeneity. QoS management is further complicated by synchronizations in parallel applications that couple thread progress, making independent per-core DVFS decisions ineffective and thread-level proxies (e.g., IPS/IPC) unreliable indicators of delivered service. This paper proposes a novel heartbeat-driven QoS scheduler that uses application heartbeats as explicit progress feedback and quantifies cross-thread progress imbalance using the coefficient of variation of per-thread inter-heartbeat intervals. At each scheduling epoch, the controller computes (i) a differential "rate harmonization" DVFS action that reduces progress dispersion across threads and (ii) a uniform QoS DVFS action that regulates the global heartbeat rate to a minimum target. We evaluate our approach using the *HotSniper* many-core simulator. Results show that our harmonization-based controller meets the QoS target more robustly and reduces average energy consumption by 43.6% relative to comparable baselines, demonstrating that synchronization-aware progress harmonization is useful for energy-efficient QoS on S-NUCA many-cores.**

***Index Terms*—Efficient Computing, Computer Systems**

I. Introduction

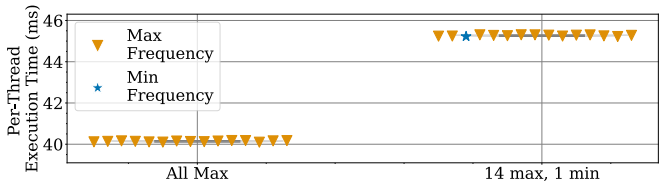
Ensuring quality of service (QoS) is a central challenge in modern many-core systems, where users and applications increasingly demand predictable performance under tight energy budgets. System designers often monitor thread-level indicators such as instructions per second (IPS) or instructions per cycle (IPC) to guide control decisions. However, these metrics frequently diverge from the service targets that matter to software. For example, a video game renderer can maintain a stable IPS/IPC while still producing an unstable frame rate because the application's progress depends on end-to-end timing and pipeline dynamics rather than raw instruction execution. Even worse, busy-wait synchronization (e.g., spin locks) can inflate IPS while producing zero useful work, making IPS appear "healthy" precisely when the application makes no progress. As a result, a stable IPS does not guarantee that an application meets its service objective, whether that objective is frame rate, latency, throughput, or deadline compliance.

Application heartbeats [1] address this mismatch by providing an explicit application-level progress signal that directly encodes the delivered service rate. In contrast to thread-centric counters, heartbeats measure what the software actually does, enabling the system to reason about QoS in the same terms that users experience. This distinction becomes crucial in many-core architectures with shared resources that introduce performance asymmetry across cores. In S-NUCA many-cores, last-level caches (LLC) remain logically shared but are physically distributed, so access latency varies with on-chip network hops and the mapping of threads to cores. This non-uniformity creates inherent heterogeneity, such that two identical cores running the same code can exhibit different effective memory-hierarchy latencies and, therefore, different progress rates. Without explicitly accounting for this heterogeneity, control policies can overreact to misleading thread-level behavior or fail to correct the true QoS bottlenecks.

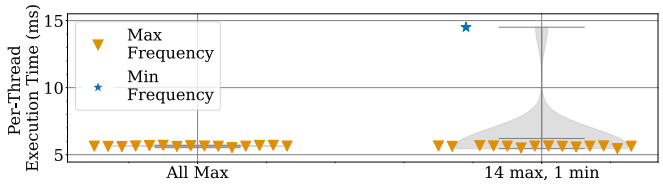
At the same time, the system's most effective control knobs, dynamic voltage and frequency scaling (DVFS), thread migration, and related mechanisms, operate at the per-core or per-thread level. This creates a fundamental control gap: the QoS signal is application-level, whereas the actuators are at the core level. Synchronization further complicates the problem because it couples thread progress and can hamper independent tuning. In synchronized workloads that contain barriers or locks, overall progress is limited by the slowest critical-path thread. Therefore, per-core adjustments must be coordinated to avoid amplifying imbalance or stalling the entire application. In unsynchronized workloads, progress aggregates across threads, so heterogeneity directly distorts QoS, and targeted per-core tuning can provide meaningful gains. Because real workloads vary in synchronization, from tightly synchronized to largely independent, an effective strategy must handle both extremes and the mixed cases between them.

This paper argues for a control strategy that treats heterogeneity at the individual core-level while maintaining QoS and optimizing energy at the application level. Prior efforts have largely relied on thread-level metrics rather than application-level progress signals [2], [3], while approaches that do use application-level metrics often ignore how synchronization changes the relationship between per-core tuning and end-to-end QoS [4]. To address these gaps, we propose a scheduling algorithm that applies a rate-harmonization technique to counteract heterogeneity and uses DVFS to maintain the required QoS while improving energy efficiency.

As a motivational example, Fig. 1 illustrates this mismatch between per-core control and actual progress of an application using per-thread wall-clock completion times for two 15-



(a) `canneal`



(b) `blackscholes`

Fig. 1: Per-thread execution times for a synchronized workload (`canneal`) and an independent workload (`blackscholes`). In the synchronized case, thread progress is strongly coupled, where throttling a single thread forces all threads to complete together. This indicates that effective management requires coordinated (typically uniform) frequency selection across threads rather than independent per-core tuning.

thread *PARSEC* [5] workloads: `canneal`, which is synchronized, and `blackscholes`, which is largely independent. For each workload, we use two DVFS configurations: (i) all threads at maximum frequency, (ii) one thread at minimum frequency and the remaining 14 at maximum. This controlled experiment exposes how synchronization determines whether per-thread tuning yields predictable application behavior.

The results sharply separate the two workload classes. In `canneal`, slowing down a single thread increases the measured completion time of every thread, even though only one core changes frequency. The faster threads simply spend more time waiting at locks/barriers, so the workload's progress collapses based on the rate of the slowest critical-path thread. In contrast, `blackscholes` exhibits mostly independent thread execution. Changing one thread's frequency primarily perturbs that thread's completion time, while other threads remain relatively stable. These observations show why naively controlling threads independently fails in tightly synchronized workloads and that per-core knobs must act in a coordinated manner to avoid creating waiting-induced inefficiency. At the same time, even independent workloads eventually synchronize at process termination, so running threads at mismatched rates creates end-of-execution slack. Rate harmonization can reclaim this slack by intentionally slowing early finishers to align completion times, reducing wasted waiting without increasing overall makespan.

II. Background and Related Work

In this section, we review the architectural and runtime foundations that motivate synchronization-aware QoS control in S-NUCA many-cores. We explain how S-NUCA's physically distributed LLC creates core-dependent latency heterogeneity and why application heartbeats provide a progress signal that better tracks delivered service than thread-level counters. We then connect these concepts to prior QoS scheduling, synchronization-aware power management, and NUCA placement work to clarify the remaining gap that our rate-harmonization strategy targets.

A. Static Non-Uniform Cache Access (S-NUCA) Architecture

Non-Uniform Cache Access (NUCA) many-core designs organize the last-level cache (LLC) as a set of banks that are *physically distributed* across the chip while remaining *logically shared* by all cores [6]. In Static-NUCA (S-NUCA), the mapping from memory addresses to LLC banks is fixed at design time by statically interleaving addresses across banks. Compared to operating-system-managed approaches such as Dynamic-NUCA (D-NUCA), this static policy offers less flexibility during execution; however, it enables a low-overhead implementation that can be realized largely in hardware [7].

To illustrate, consider a 64-core many-core arranged as an 8×8 mesh. A typical S-NUCA organization pairs each core with a colocated LLC slice, producing 64 cache banks distributed over the grid. When a core issues an LLC request, the request must traverse the on-chip interconnect to reach the bank that holds the corresponding cache line. Consequently, LLC access time depends on the communication distance between the requesting core and the destination cache bank and on the routing delay through the Network-on-Chip (NoC).

A common way to model this distance is the Manhattan hop count on the mesh. In S-NUCA, static address interleaving causes threads to touch many different banks over time, often with approximately uniform probability across the LLC slices. Under this assumption, the average LLC access latency experienced by a given core correlates with its *Average Manhattan Distance (AMD)* to all cache banks [8]. Cores closer to the chip center tend to have smaller AMD and therefore lower average cache access latency, whereas edge and corner cores have larger AMD and incur higher average latency.

B. Application Heartbeats

The *Application Heartbeats* framework [1] provides a lightweight, application-defined notion of progress that directly reflects useful work rather than raw instruction retirement. The programmer inserts a heartbeat at a logical point where each occurrence corresponds to a comparable unit of work (e.g., one loop iteration, one processed request, or one simulation step). At runtime, the system logs the timestamp of each heartbeat event and uses these timestamps to derive a *heart rate* (HR) metric that tracks delivered progress over time. Unlike IPS, which can remain high even during spin waiting or other non-productive execution, HR increases only when the application completes work units and therefore serves as a more faithful signal for QoS-oriented control.

We compute HR using a sliding window over the most recent heartbeat timestamps to smooth transient jitter while preserving responsiveness. Let $t(k)$ denote the timestamp of

the k-th heartbeat and w the window size (in number of heartbeats). We estimate HR at heartbeat index k as

$$\mathrm{HR}(k) = \frac{w}{t(k) - t(k-w)} \tag{1}$$

This formula reports the average completion rate over the last w work units, attenuating noise in individual timestamps and enabling stable feedback for DVFS and scheduling decisions.

In parallel programs, worker threads typically emit heartbeats at the same logical progress point (e.g., after completing an iteration or task). The relative alignment of heartbeat intervals across threads therefore reveals the degree of synchronization or coupling. A tightly synchronized workload tends to keep equivalent heartbeat interval lengths closely aligned, whereas loosely coupled workloads allow for greater divergence. We assess synchronization by comparing inter-heartbeat intervals and the time skew between threads. Because observed skew can arise from both true synchronizations and external perturbations (e.g., OS noise), we evaluate these measures over short windows to reduce sensitivity to noise.

C. Related Work

Prior work has demonstrated the value of exposing application-defined progress to runtime systems. *Application Heartbeats*, introduced by Hoffmann et al. [1], provides a lightweight interface for applications to emit progress events and specify performance goals. Subsequent studies [9], [10] leverage heartbeat feedback to meet performance objectives while reducing power, including approaches that tune voltage/frequency to satisfy progress targets with minimal energy. Muthukaruppan et al. [11] have explored hierarchical control for heterogeneous processors by coordinating multiple PID-style controllers to keep task heart rate within a target band on ARM big.LITTLE under power constraints. Collectively, these efforts establish heartbeats as an effective control signal, but they largely treat the feedback as an application-level scalar and do not explicitly examine how per-core actuation interacts with many-core heterogeneity and synchronization coupling to determine end-to-end QoS.

A complementary line of work studies the energy cost of synchronization and waiting. Techniques such as those of Li et al. [12] reduce spin-wait waste by predicting barrier stall time and transitioning early arriving threads into low-power states, while other efforts identify common synchronization patterns and apply pattern-specific DVFS coordination to reduce spinning and blocking [13]. A study by Cesarini et al. [14] on communication slack shows that busy waiting during communication phases can dominate wasted cycles, and that targeted runtime policies can recover energy with small overhead. Several studies explicitly argue that DVFS prediction and control must account for synchronization dependencies, since the slowest thread often gates global progress [15]. These synchronization-aware methods primarily target local waiting inefficiencies (sleeping at barriers or during communication) or focus on predicting the performance impact of synchronization. In contrast, our approach uses application progress feedback to harmonize per-thread execution rates, reducing progress dispersion before threads converge at barriers/locks or at join, and couples this harmonization with QoS regulation under per-core DVFS actuation.

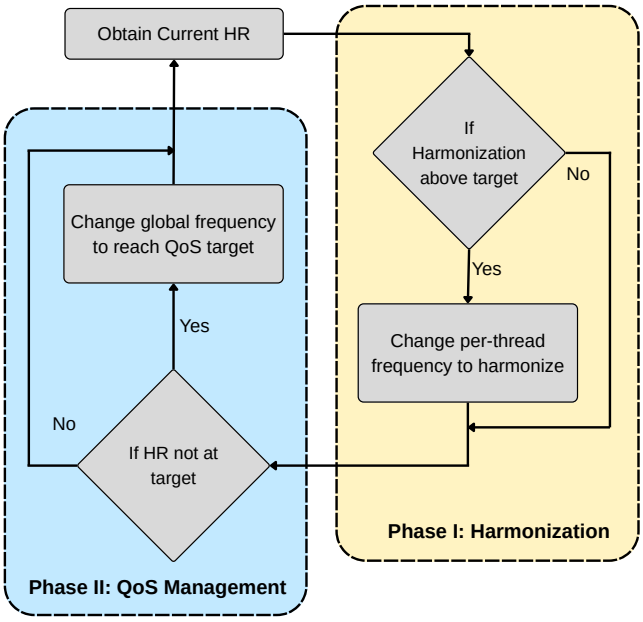


Fig. 2: Overview of the control loop

Many-core S-NUCA management introduces an additional axis of heterogeneity: non-uniform LLC access latency that depends on placement and on-chip distance. Rapp et al. propose PCMig [16], which evaluates candidate thread migrations in S-NUCA using models that predict IPS impact and selects placements to improve performance under latency and power constraints. Their later work [2] refines these predictors using lightweight neural and analytical models. Other policies [17] also exploit the S-NUCA structure to mitigate placement-induced variability. These systems largely optimize placement and performance using low-level proxies such as IPS, which can diverge from application-level service. The closest work is Wasala et al. [4], who regulate application QoS using heart-rate feedback, DVFS, and migration. However, their control design does not explicitly model or mitigate synchronization-induced progress coupling, which can cause per-core actions to translate nonlinearly into end-to-end QoS. Our contribution builds on heartbeat-based regulation and adds synchronization-aware rate harmonization to address memory-system heterogeneity and coupled thread progress in S-NUCA many-cores.

III. Algorithm

We implement the scheduler as a single closed-loop controller (Fig. 2) that uses heartbeats as its primary feedback signal. The controller addresses both harmonization and QoS management objectives concurrently within the same iteration.

Heartbeats provide a per-thread notion of forward progress that thread-level counters cannot reliably capture. Each thread emits a heartbeat when it completes an application-defined work unit; the k-th heartbeat therefore marks completion of

the k-th unit by that thread. We measure HR at the granularity of a control epoch e, where the epoch duration is fixed by the controller configuration and is independent of the heartbeat index k. We require a cross-thread metric to define the dispersion for harmonization. We first calculate the latest heartbeat interval over a sliding window for each thread. For a given scheduling epoch e, let $latest(e)$ be the largest heartbeat index that each thread has reached at any given time, and let $t_i(k)$ denote the timestamp of the k-th heartbeat observed for thread i. We derive a per-thread inter-heartbeat interval:

$$\Delta t_i(e) = t_i(latest(e)) - t_i(latest(e) - W_{harm}) \quad (2)$$

where W_{harm} is the window used for harmonization. This measures how long thread i took to complete its latest W_{harm} units of work at each scheduling epoch. These intervals implicitly include stalls due to non-uniform cache access, on-chip network contention, and synchronization delays, making them a direct proxy for effective per-thread progress rate under the current mapping and DVFS configuration.

To quantify how well threads advance together, we compute a dispersion metric across the set $\{\Delta t_i(e)\}_{i=1}^{n}$ (i.e. across all threads at any given epoch). We first compute the cross-thread mean ($\mu_\Delta(e)$) and standard deviation ($\sigma_\Delta(e)$) of inter-heartbeat intervals.

$$\mu_\Delta(e) = \frac{1}{n}\sum_{i=1}^{n} \Delta t_i(e) \quad (3)$$

$$\sigma_\Delta(e) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} \left(\Delta t_i(e) - \mu_\Delta(e)\right)^2} \quad (4)$$

We then define the coefficient of variation (CV) as our harmonization metric.

$$CV_\Delta(e) = \frac{\sigma_\Delta(e)}{\mu_\Delta(e)} \quad (5)$$

An ideally synchronized workload yields identical inter-heartbeat intervals across threads, thereby achieving $CV_\Delta(e) = 0$. In practice, OS interference, measurement noise, and minor phase variability introduce small deviations even under tight synchronization, so we define a tolerance ϵ and consider the workload sufficiently harmonized when $CV_\Delta(e) \leq \epsilon$. Within each control iteration e, the controller computes two coupled adjustments from heartbeat data: a differential action that reduces rate dispersion across threads (harmonization) and a uniform action that regulates application QoS. We harmonize by directly driving each thread's latest inter-heartbeat interval $\Delta t_i(e)$ toward the cross-thread mean $\mu_\Delta(e)$. Specifically, we compute a normalized deviation for each thread,

$$\beta_i(e) = \frac{\Delta t_i(e) - \mu_\Delta(e)}{\mu_\Delta(e)} \quad (6)$$

and the frequency change required to harmonize each thread,

$$\Delta f_i^{harm}(e) = HarmInc \cdot n \cdot \beta_i(e) \quad (7)$$

where n is the number of threads and $HarmInc$ is a tunable harmonization gain. This update increases frequency for slower-than-average threads and decreases frequency for faster-than-average threads, thereby pulling each thread's progression rate closer to the mean. We apply $\Delta f_i^{harm}(e)$ only when $CV_\Delta(e) > \epsilon$; otherwise, we set $\Delta f_i^{harm}(e) = 0$ to avoid injecting unnecessary per-thread perturbations once the workload is sufficiently harmonized. Reapplying this policy each iteration iteratively reduces dispersion for weakly synchronized and independent workloads until $CV_\Delta(e) \leq \epsilon$ or DVFS bounds prevent further adjustment, mitigating the inherent heterogeneity of the S-NUCA many-core.

In addition to the dispersion signal used for harmonization, the controller continuously regulates QoS using the application heart rate HR. At each epoch, the controller estimates the current heart rate using a sliding window over the most recent W_{qos} heartbeat observations across all threads using Equation (1). This windowed estimate gives the average global heart rate of the application.

The controller computes a uniform DVFS correction from this windowed QoS estimate and applies it to all participating cores. This uniform action runs concurrently with the harmonization action: harmonization redistributes per-core frequencies to reduce rate dispersion, while the QoS controller shifts the overall frequency level up or down to enforce the service constraint HR $\geq$ $\text{HR}_{\text{target}}$ with good energy efficiency. At epoch e, the controller computes a normalized error term

$$\alpha(e) = \frac{(\text{HR}_{\text{target}}) - \text{HR}(e)}{\text{HR}_{\text{target}}} \quad (8)$$

We introduce a deadband at 5% above the HR target to improve robustness. When HR(e) is in between $\text{HR}_{\text{target}}$ and $\text{HR}_{\text{target}}$ + 5%, $\alpha(e)$ is set to 0. We then update the global frequency level using an incremental gain $QoSInc$:

$$\Delta f_{\text{qos}}(e) = QoSInc \cdot n \cdot \alpha(e) \quad (9)$$

where n is the number of threads. When HR(e) drops below the target, $\alpha(e)$ is greater than 0 and the controller increases frequency; when HR(e) exceeds the target plus the deadband, $\alpha(e)$ is smaller than 0 and the controller decreases frequency to save energy. The controller applies this update uniformly across all participating cores, while the harmonization component simultaneously adjusts relative per-core frequencies.

$$f_i(e+1) = f_i(e) + \Delta f_{\text{qos}}(e) + \Delta f_i^{\text{harm}}(e) \quad (10)$$

In order to reduce the chance of the system oscillating due to the concurrent structure, we explicitly setup $harmInc$ to be much smaller than $QoSInc$, ensuring that any change in the frequency will be minor and adjusted in the following iteration. This maintains application-level QoS without reintroducing imbalance, and it enables the controller to treat heterogeneity correction and service regulation as complementary parts of a single stable feedback policy.

IV. Evaluation

This section presents an empirical evaluation of our harmonization-based QoS management policy. We conduct all experiments using the heartbeat-integrated HotSniper [18] simulator, which models many-core execution and exposes application heartbeats as progress signals. Heartbeat-based control requires the programmer to place heartbeat calls at code locations that faithfully represent execution progress, such as the same logical point in the main loop of each worker thread. Hoffmann et al. showed that such instrumentation is practical [1] in the PARSEC benchmark suite; however, many benchmarks still emit too few heartbeats to support a real-time feedback loop, especially when the controller requires thread-level progress measurements. Prior work on heartbeat-driven QoS for S-NUCA systems used three workloads that represent processor-intensive, memory-intensive, and moderate execution behavior: `blackscholes`, `canneal`, and `dedup`, respectively [4]. That study measured heart rate only at the global application level. In contrast, our harmonization controller requires enough heartbeats from each thread to estimate per-thread progress dispersion and adjust DVFS decisions online. We therefore evaluate `blackscholes`, `canneal`, and `swaptions` from the PARSEC benchmark suite using `simsmall` inputs. This set preserves coverage of diverse workload behavior while providing sufficient per-thread heartbeat activity for real-time synchronization-aware harmonization.

For each benchmark, we first identify an achievable heart rate (HR) envelope by simulating best- and worst-case operating conditions. We obtain the maximum-performance point by running at the highest frequency while mapping threads to centrally located cores, and we obtain the minimum-performance point by running at the lowest frequency with threads placed near the edges of the chip. We then randomly select target HR values within this per-benchmark range. These values define the target operating ranges used in our QoS experiments. We model a 64-core out-of-order S-NUCA system arranged as an 8×8 mesh NoC. Each core includes private 16 KB L1 instruction (L1-I) and 16 KB L1 data (L1-D) caches. The shared last-level cache (LLC) has 8 MB total capacity and is partitioned into 64 banks of 128 KB each, colocated with the cores. The NoC incurs 1.5 ns per hop, corresponding to 6 cycles at 4.0 GHz, and uses 256-bit links.

We compare our policy against two representative state-of-the-art schedulers: PCMig [2] and EEQoS [4]. EEQoS follows a reactive control strategy that maintains the observed heart rate within a specified target range by applying DVFS and thread migrations. While this design captures application-level QoS through heartbeats, it treats threads independently and does not explicitly account for synchronization effects that can couple thread progress and amplify waiting. In contrast, PCMig targets NUCA systems with a learning-based model that predicts the performance impact of candidate frequency changes and migrations. PCMig optimizes for response time using thread-level indicators, specifically IPS, rather than heartbeat-derived progress, and therefore remains agnostic to application-defined work units and heartbeat semantics. Together, these baselines allow us to isolate the benefits of (i) heartbeat-driven QoS control versus IPS-centric prediction, and (ii) synchronization-aware harmonization versus reactive QoS management without coupling awareness.

A. QoS Management Evaluation

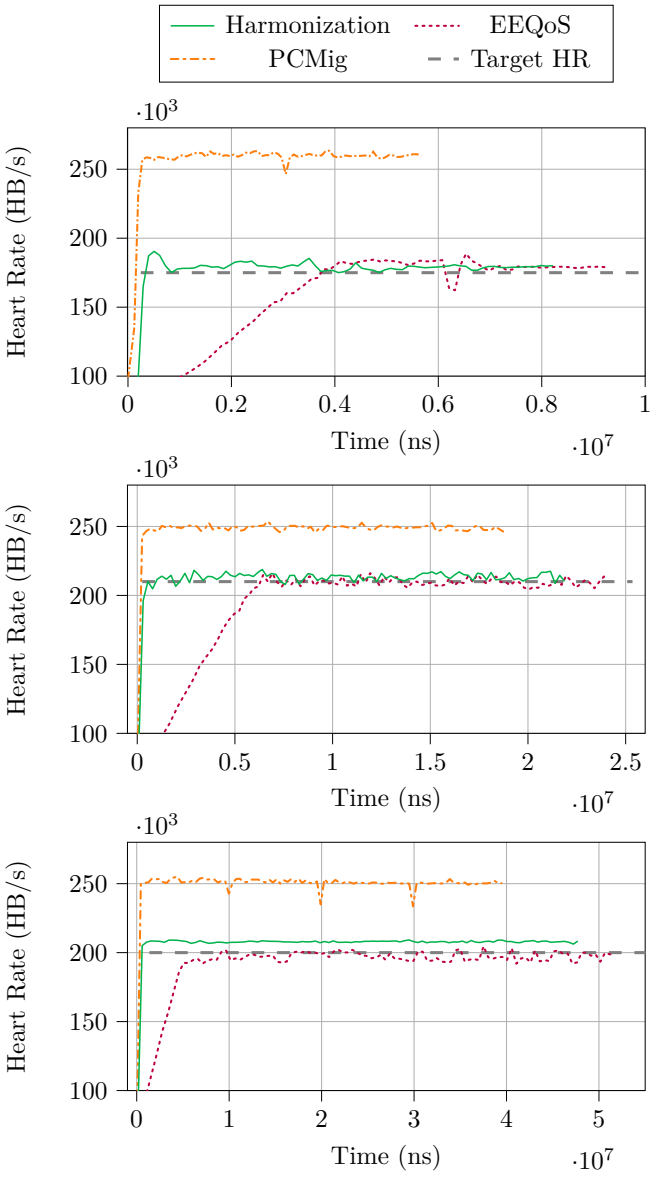


Fig. 3: Performance of our policy compared to EEQoS and PCMiG. Each application uses 15 parallel threads. (top: `blackscholes`, middle: `swaptions`, bottom: `canneal`)

Fig. 3 shows the global heart rate (HR) achieved by the three policies over time for the selected *PARSEC* workloads, using 15 parallel threads per execution. Because the primary objective of QoS management is to satisfy a target HR requirement, we evaluate each policy along three dimensions: (i) how quickly it drives HR into the target region after

startup and phase changes, (ii) how effectively it keeps HR above the minimum acceptable level once stabilized, and (iii) how much oscillation it introduces around the target after convergence. These criteria capture both responsiveness and stability, which jointly determine whether the system delivers predictable service without excessive over-provisioning.

Across all three benchmarks, our harmonization-based policy reaches the target faster than EEQoS and converges in roughly the same time as PCMig. However, the stabilized behavior differs substantially. PCMig optimizes for performance (response time) using IPS rather than heartbeat targets, and it does not explicitly regulate HR to a target range. Consequently, it often remains above the target band, satisfying the requirement but doing so incidentally rather than by design. In contrast, EEQoS is formulated around maintaining HR within a specified range rather than enforcing a minimum target. To compare fairly against our minimum-target formulation, we configure EEQoS using the upper limit of our policy's deadband; this choice increases controller aggressiveness and induces oscillations that repeatedly drive HR below the minimum acceptable target. This behavior highlights the importance of jointly addressing stability and coupling, a reactive range-tracking policy can satisfy the target on average while still violating QoS intermittently due to sustained undershoot during oscillatory corrections.

B. Energy optimization

Fig. 4 compares the total energy consumed by the three schedulers while executing each benchmark under its assigned QoS target. Our harmonization-based policy consistently achieves the lowest energy across `blackscholes`, `swaptions`, and `canneal`, indicating that it meets the required service level with less over-provisioning than the baselines. The savings are most pronounced for `swaptions`, where EEQoS expends substantially more energy due to aggressive corrective actions, while PCMig remains above the QoS requirement because it optimizes for performance (IPS/response time) rather than targeting a minimum heart rate. In contrast, our controller uses heartbeat feedback to avoid sustained operation above the necessary rate and leverages harmonization to reduce slack and waiting, enabling lower frequencies without sacrificing QoS. For the tightly synchronized `canneal`, the energy gap narrows, synchronization constrains per-core freedom and forces the system toward a more uniform operating point, yet our policy still provides the best energy outcome by minimizing oscillatory DVFS adjustments and eliminating unnecessary headroom once the heart rate stabilizes.

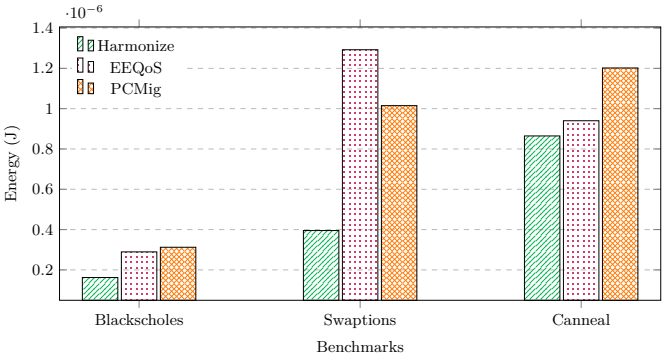


Fig. 4: Energy consumption of different policies. Our approach produces better energy consumption in all cases compared to the baselines.

V. Conclusion

In S-NUCA many-cores, physically distributed shared caches introduce core-dependent latency heterogeneity, while synchronizations can couple thread progress and distort how per-core actions translate to end-to-end QoS. To address this control gap, we proposed a heartbeat-driven scheduler that explicitly accounts for heterogeneity by harmonizing per-thread progress rates before applying QoS regulation. Our controller uses application heartbeats as a direct progress signal and computes both a differential harmonization adjustment (to reduce cross-thread dispersion) and a uniform QoS adjustment (to maintain a minimum target heart rate) in each epoch. This combined policy counteracts S-NUCA-induced heterogeneity while avoiding the pitfalls of independent per-thread tuning under strong coupling. Across *PARSEC* workloads that we use for evaluation, our harmonization-based policy reaches the QoS target faster than EEQoS and exhibits improved stability relative to range-tracking behavior that can undershoot the minimum target. It also consistently achieves the lowest energy among the evaluated schedulers by reducing waiting/slack and avoiding sustained operation above the required service rate.

References

[1] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal, "Application heartbeats: a generic interface for specifying program performance and goals in autonomous computing environments," in *ICAC*, 2010, pp. 79–88.

[2] M. Rapp, A. Pathania, T. Mitra, and J. Henkel, "Neural network-based performance prediction for task migration on S-NUCA many-cores," *IEEE Transactions on Computers*, 2020.

[3] M. Rapp, H. Khdr, N. Krohmer, and J. Henkel, "NPU-accelerated imitation learning for thermal optimization of QoS-constrained heterogeneous multi-cores," *TODAES*, 2018.

[4] S. M. Wasala *et al.*, "Energy-efficient QoS-aware scheduling for S-NUCA many-cores," in *2025 26th International Symposium on Quality Electronic Design (ISQED)*. IEEE, 2025, pp. 1–8.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, 2008, pp. 72–81.

[6] R. Balasubramonian, N. P. Jouppi, and N. Muralimanohar, *Multi-core cache hierarchies*. Morgan & Claypool Publishers, 2011.

[7] H. Kim, P. Ghoshal, B. Grot, P. V. Gratz, and D. A. Jiménez, "Reducing network-on-chip energy consumption through spatial locality speculation," in *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip*, 2011, pp. 233–240.

[8] A. Pathania and J. Henkel, "Task scheduling for many-cores with S-NUCA caches," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 557–562.

[9] H. Hoffmann *et al.*, "Dynamic knobs for responsive power-aware computing," *ACM SIGARCH computer architecture news*, vol. 39, no. 1, pp. 199–212, 2011.

[10] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal, "A generalized software framework for accurate and efficient management of performance goals," in *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*. IEEE, 2013, pp. 1–10.

[11] T. S. Muthukaruppan, M. Pricopi, V. Venkataramani, T. Mitra, and S. Vishin, "Hierarchical power management for asymmetric multi-core in dark silicon era," in *Proceedings of the 50th Annual Design Automation Conference*, 2013, pp. 1–9.

[12] J. Li, J. F. Martinez, and M. C. Huang, "The thrifty barrier: Energy-aware synchronization in shared-memory multiprocessors," in *10th International Symposium on High Performance Computer Architecture (HPCA'04)*. IEEE, 2004, pp. 14–23.

[13] Y. D. Liu, "Energy-efficient synchronization through program patterns," in *2012 First International Workshop on Green and Sustainable Software (GREENS)*. IEEE, 2012, pp. 35–40.

[14] D. Cesarini *et al.*, "Countdown slack: A run-time library to reduce energy footprint in large-scale MPI applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 11, pp. 2696–2709, 2020.

[15] S. Akram, J. B. Sartor, and L. Eeckhout, "DVFS performance prediction for managed multithreaded applications," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2016, pp. 12–23.

[16] M. Rapp, A. Pathania, T. Mitra, and J. Henkel, "Prediction-based task migration on S-NUCA many-cores," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 1579–1582.

[17] Y. Shen, S. Niknam, A. Pathania, and A. D. Pimentel, "Thermal management for s-nuca many-cores via synchronous thread rotations," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.

[18] A. Pathania and J. Henkel, "HotSniper: Sniper-based toolchain for many-core thermal simulations in open systems," *IEEE Embedded Systems Letters*, vol. 11, no. 2, pp. 54–57, 2018.