KarmaPM: Reward-Driven Power Manager

Sunil Kumar[0000−0002−0891−7847] and Vivek Kumar✉[0000−0003−3042−4202]

IIIT-Delhi, India
{sunilk,vivekk}@iiitd.ac.in

Abstract. Hardware overprovisioning is a widely used technique to improve the average power utilization of computing systems by capping the processor's power consumption. However, applying a uniform power cap across multiprocessor system sockets can significantly impact co-running applications due to workload variations. This paper introduces *KarmaPM*, a novel power management library for co-running applications on multiprocessor systems, independent of the parallel programming model, based on application power donation phases. KarmaPM dynamically redistributes power bidirectionally across the sockets to improve overall system throughput for co-running applications while maintaining fairness between them. KarmaPM periodically profiles the CPU utilization of each application. When it detects an application underutilizing its CPU resources, it donates the surplus power from this donor application's sockets to the other sockets (receivers), exhibiting high CPU utilization. When the donor application enters a high CPU utilization phase, KarmaPM employs a reward power scheme that rewards the donor application by returning a portion of the power transferred to the receiver sockets. We evaluated KarmaPM across various exascale proxy application mixes and power caps on a four-socket, 72-core Intel Cooper Lake processor. Our results show that KarmaPM improved the system throughput (geometric mean) by 13.2% at a lower power cap and 6.6% at a higher power cap. Additionally, KarmaPM delivered improvements of 12.5% and 4.4% in system throughput (geomean) compared to an existing power manager at these respective power caps.

Keywords: Hardware overprovisioning · mulitprocessor · system throughput · co-running applications

1 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 enables concurrent execution of multiple applications on multiprocessor systems. However, varying power requirements among co-running applications necessitate limiting processor consumption to reduce the carbon footprint. To manage rising power demands, data centres and supercomputers commonly employ hardware overprovisioning for capping power usage at some processors below their Thermal Design Power

(TDP) limit [11] while reallocating unused power to support additional computing systems (e.g., GPUs, and nodes in the HPC cluster). Modern processors support hardware overprovisioning through power capping (PCAP), which enforces user-defined limits by throttling frequency at the hardware level. However, operating a processor under a PCAP can significantly degrade the application's performance [15]. Parallel applications often exhibit intermittent phases of low CPU utilization during execution, referred to as slacks. These phases occur when only a subset of the allocated cores actively execute application threads while the remaining cores remain idle. Slacks typically arise during sequential execution, I/O operations, or synchronization barriers, in contrast to parallel regions where CPU resources are fully utilized. A well-studied approach to improving the performance of such applications running under PCAP in an HPC cluster is to use power managers (PMs) that redistribute surplus power from slack-experiencing nodes to busy nodes [9, 5, 10, 4]. However, a common limitation of these solutions is that they do not focus on inter-socket power scheduling within a multiprocessor system. Additionally, they fail to account for fairness in the system by not rewarding the power donor, as power is transferred unidirectionally from processors running under slack to those executing parallel regions.

This paper proposes KarmaPM, a novel reward-based power manager designed to enhance the fairness and throughput of co-running applications in a multi-socket system under a PCAP based on the application's power donation *karmas*[1]. KarmaPM operates independently as a lightweight daemon process alongside the co-running applications and requires no prior knowledge of application characteristics. KarmaPM periodically monitors the CPU utilization of applications at each socket. When it identifies an application underutilizing its CPU resources, it reallocates surplus power from this donor application's sockets to other receiver sockets that fully utilize its CPU resources. KarmaPM records the duration the donor application has provided power to the receiver sockets. When the donor application resumes fully utilizing its CPU resources, KarmaPM rewards that application by returning a fraction of the total power previously donated to the receiver sockets. The power transfer between sockets is achieved by reducing the PCAP at the donor socket while increasing the PCAP at the receiver socket by the same amount, ensuring that the overall power consumption remains within the user-set PCAP limits.

In summary, this paper makes the following contributions:

- KarmaPM, a library-based power management solution for multiprocessor systems that enhances fairness and throughput for co-running parallel applications on a power-capped multiprocessor server while adhering to the system power budget.
- A lightweight profiler that monitors CPU utilization across each socket, facilitating the transfer of excess power from underutilized sockets to those fully utilizing their CPU resources.

[1] Sanskrit word referring to the sum of somebody's good and bad actions in one of their lives. KarmaPM only pays attention to good karmas.

– A novel power reward mechanism designed to promote fairness and throughput by compensating power-donating sockets with a portion of the donated power as they transition from low to high CPU utilization phases.
– An evaluation of KarmaPM on a quad-socket 72-core Intel Xeon processor using diverse mixes of exascale proxy applications [2] and various PCAPs, demonstrating that KarmaPM can substantially improve the system throughput and application fairness.

2 Related Work

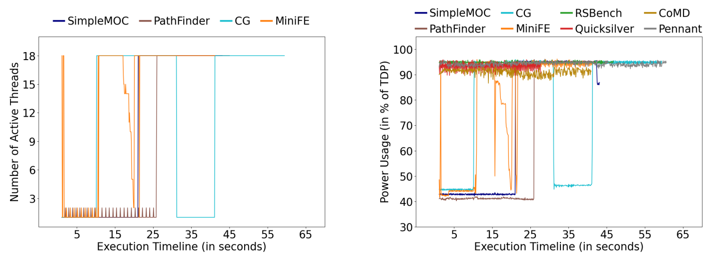
Hardware overprovisioning [11] is widely used to reduce the increasing power consumption cost and carbon footprint of data centres and supercomputers. It has been used extensively both at the cluster and single server levels. Broadly, these solutions can be divided into two categories: one that aims to improve the performance of applications running at cluster level using inter-node power scheduling, and the other that seeks to improve the performance of applications running at server level. Patki et al. [11] proposed an approach for cluster-level power budgeting by configuring a fixed power budget at each node. However, applying uniform power allocation at each node impacts the application's performance when its power demand increases at some nodes. Such issues are common in MPI applications where some ranks could experience slack while others are busy (e.g., due to barrier synchronization [5] or manufacturing variability in processors [7]). Several approaches exist for power scheduling across nodes running different MPI applications for improving system throughput [16, 9, 4] and application level fairness [10]. Improving the system throughput at a multiprocessor server has been explored by configuring processor frequency for a single running application [14, 6] or combining it along with tuning the thread count for co-running applications [15].

Cluster-based power managers transfer power unidirectionally, redistributing it from donor to receiver nodes without ensuring fairness through rewarding donors. Existing server-level solutions optimize processor resources to minimize performance loss. However, none of the approaches *transparently* enhances system throughput and fairness for co-running applications on a multiprocessor system using power scheduling. Recently, Costero et al. [3] proposed a dynamic power distribution solution for such systems, but it requires modifications to the OpenMP runtime and is limited to co-running applications using OpenMP tasking pragma. KarmaPM bridges this gap with a programming model-oblivious, reward-based inter-socket power scheduling approach that dynamically adjusts power allocation based on an application's instantaneous usage, requiring no prior knowledge of input applications. A power rewarding scheme was recently explored for the Fugaku supercomputer, where applications with low power usage earn tokens to reduce their wait time in the job queue during relaunch [13].

3 Experimental Methodology

We first describe our experimental methodology, followed by a motivating analysis for KarmaPM. We used eight exascale proxy applications for our experimental evaluations [2]. These applications and input parameters are: (1) **SimpleMOC** (n_azimuthal=64, cai=3, fai=3, decomp_assemblies_ax=5), (2) **PathFinder** (input_file=10kx750.adj_list), (3) **CG** (class=C), (4) **MiniFE** (nx=256, ny=300, nz=512, iterations=60), (5) **RSBench** (p=1000000), (6) **Quicksilver** (input_file=Coral2_P1.inp, n=200000), (7) **Pennant** (input_file=leblancbig.pnt, meshparams=120x1080, tstop=10.0), (8) **CoMD** (n=40x40x40, nSteps=800). We choose different parallel programming models in these applications as discussed in Section 6 to demonstrate the parallel runtime obliviousness in KarmaPM. We did not modify the applications otherwise, but we changed the default parameters to control the execution time on our machine. We used GNU compiler version 10.3 with the -O3 optimization to compile the applications. 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). Hyperthreading was disabled, and turbo boost was enabled. Our machine had 512GB of RAM and Ubuntu 20.04.5 LTS operating system (OS) with Linux Kernel 5.4. To set the socket-level power cap (PCAP), we used the Intel RAPL (Running Average Power Limit) interface by writing to the `MSR_PKG_POWER_LIMIT` register for each socket. Our experimental evaluations used three different PCAPs: 83 Watts (55% of TDP), 98 Watts (65% of TDP), and 112 Watts (75% of TDP). A PCAP setting of 150 Watts (the TDP for each socket) indicates that each socket's PCAP is individually set to 150 Watts, resulting in a total system-level PCAP of 600 Watts. We preserved the same system and application settings across all PCAPs.

4 Motivating Analysis



(a) Periods of slack resulting from sequential phases

(b) Power usage correlates with the presence of slack

Fig. 1. Execution timeline of applications at TDP

This section explains the rationale behind KarmaPM's design by analyzing our selected application's CPU utilization and power consumption patterns during execution. We conducted these experiments by running each application on a single socket of our quad-socket machine. Figure 1(a) presents CPU utilization trends for SimpleMoC, Pathfinder, MiniFE, and CG, demonstrating periods of slack due to sequential execution phases. For CG, slack occurs twice, totalling approximately 25% of the total execution time. The other three applications initially exhibit slack, ranging from 23% to 49% of execution. The remaining four applications, RSBench, Quicksilver, Pennant, and CoMD, do not experience any slack during their execution timeline. Slack also impacts power consumption, as shown in Figure 1(b), where reductions align with the slacks in Figure 1(a)

5 Design and Implementation

The previous section highlighted that a parallel application's power consumption is influenced by its instantaneous CPU utilization. These two metrics can be measured dynamically using a lightweight online daemon profiler, eliminating the need for prior application execution data. This daemon facilitates inter-application power scheduling based on the CPU utilization of individual applications within a multiprocessor server. Our approach employs a lightweight daemon process called KarmaPM, which is launched alongside parallel applications and periodically monitors the active core count and power usage of each co-running application. Applications are launched on the server ensuring that sockets are not shared between applications, although a single application may use multiple sockets. Each application exclusively uses local socket resources such as CPU and memory. KarmaPM activates at regular intervals to minimize interference with the application's execution. No changes to applications are needed to use KarmaPM. Our insight is to initially redirect surplus power from an underutilized sockets of one application to the busier sockets of other applications to enhance overall system performance. Subsequently, when the donor application transitions to a busy state, it is rewarded with a portion of the transferred power to maintain fairness in the system.

$$P_{\text{Donated}}(X,Y) = \left(\sum_{i=1}^{k} P_{\text{Donated},i} \cdot n_i\right) / \left(\sum_{i=1}^{k} n_i\right)$$

- $P_{\text{Donated}}(\text{X}, \text{Y})$: Average power donated from App X to App Y over $\sum_{i=1}^{k} n_i$ epochs.
- $P_{\text{Donated},i}$: Power donated from x to y in the i-th epoch group (continuous phase of execution).
- n_i: Number of epochs in the i-th epoch group.
- k: Total number of epoch groups.

Fig. 2. Formula to calculate net average power donated from application X to Y

5.1 KarmaPM Daemon Loop

Algorithm 2 presents the pseudocode implementation of the KarmaPM daemon process designed for a multicore server with multiple sockets. Initially, it sets the default PCAP for each socket as specified by the user (Line 3). The daemon operates in a loop as long as at least two applications remain active (Line 5).

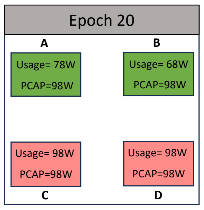
Algorithm 2: KarmaPM daemon process loop

```
1  Initialization:
2  State[$N_{Application}$], $P_{Usage}$[$N_{Application}$];
3  Set $PCAP_{Default}$ on each socket;
4  sleep(warmup duration);
5  while atleast two applications are running do
6      for application = 1 to $N_{Application}$ do
7          if application is running then
8              Store application's socket power into $P_{Usage}$[application];
9              if all cores are busy then
10                 State[application] ← CPU_Busy;
11             else
12                 State[application] ← CPU_Slack;
13             end
14         else
15             State[application] ← INACTIVE;
16             Set $PCAP_{Default}$ on application's socket;
17         end
18     end
19     KarmaPM _Policy($P_{Usage}$, State);
20     sleep ( $T_{epoch}$);
21 end
22 Set $PCAP_{Default}$ on each socket;
```

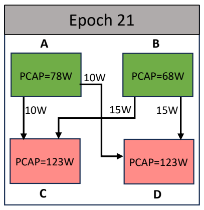
Within each iteration, it iterates over all running applications (Line 6) to measure their power consumption (Line 8) and CPU utilization (Lines 9–13) on the sockets where they are executing. If an application terminates on a socket, the daemon marks that application's state as inactive (Line 15) and resets the PCAP to the default value set by the user (Line 16). An alternative design choice would be distributing the surplus power available at an inactive socket due to its application termination among the busy applications. We did not follow it in KarmaPM implementation to ensure a fair evaluation. Subsequently, the daemon applies the application-level KarmaPM power scheduling policy (Line 19). Finally, it returns to sleep for a fixed epoch of 100ms (Line 20). Due to cold caches at the start of the execution, the KarmaPM daemon loop activates only after a warmup duration of two seconds (Line 4) to allow the system to stabilize. Before concluding its operation, the KarmaPM daemon resets the PCAP on each socket to the user-defined default value (Line 22).

5.2 KarmaPM Power Scheduling Policy

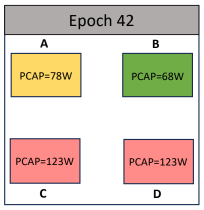
Figure 3 demonstrates the working of KarmaPM power scheduling policy, illustrated through a running example of a multiprocessor server with four sockets labelled Socket_A, Socket_B, Socket_C, and Socket_D. A single instance of the KarmaPM daemon process is launched on this server alongside applications Bench_A, Bench_B, Bench_C, and Bench_D, which are assigned to sockets Socket_A, Socket_B, Socket_C, and Socket_D, respectively. The KarmaPM process is pinned to the last core in SocketA, and the affinity of an application's threads is set to its respective sockets using the *taskset* command (the daemon process shares a core with the last thread of Bench_A). To ensure socket-local memory allocation, the `numactl` command line tool is utilized, allowing the memory pages of each application to be allocated in their respective socket-local DRAM.



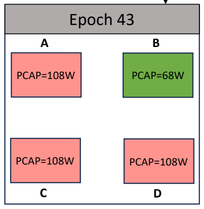
(a) Power usage at sockets A and B are 78W and 68W, respectively, due to slack, shown in green, while others in red use full power 98W.



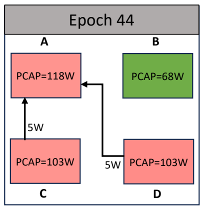
(b) PCAP of sockets A and B are decreased by 20W and 30W respectively as they donated their surplus power to other sockets equally. Hence, the PCAP of other sockets is increased by 25W.



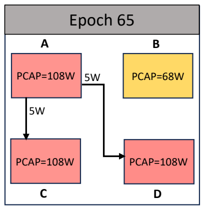
(c) After Epoch 42, socket A transitions to high CPU utilization, shown in yellow. Hence, A has no more surplus power.



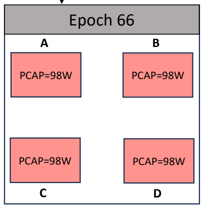
(d) Total surplus power is now 30W as only socket B has slack. Hence, it is distributed over sockets A, C and D and set their PCAP to 108W.



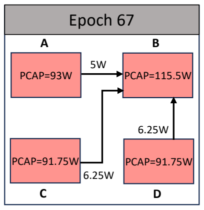
(e) Socket A is rewarded 5W from each C and D from Epoch 45 to 64 after transitioning into high CPU utilization. Hence, the PCAP of sockets A, C and D are set to 118W, 103W and 103W, respectively.



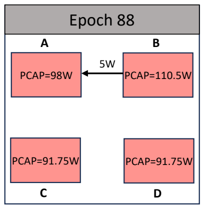
(f) At Epoch 65, socket B returns to high CPU utilization. Hence, B has no more surplus power.



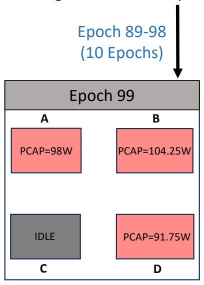
(g) All sockets are now operating under full load. Hence, PCAP is reset to default PCAP (98W) at each socket.



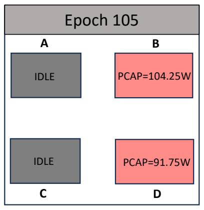
(h) Socket B is rewarded 5W from A for 20 Epochs and 6.25W from each C and D for 40 Epochs, increasing its PCAP by 17.5W. Meanwhile, the PCAP of A, C, and D decreases by 5W, 6.25W, and 6.25W, respectively.



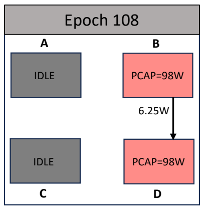
(i) After 20 Epochs, A stops rewarding B, resetting PCAP of A to 98W and reducing PCAP of B by 5W, while other sockets continue rewarding B for 20 more Epochs.



(j) At Epoch 99, socket C has stopped rewarding B as it is idle now. Hence, the PCAP of C is reset to 98W, and the PCAP of B is decreased by 6.25W.



(k) At Epoch 105, socket A has been idle now. Socket D continues giving power to B.



(l) Socket D stopped rewarding B as 40 Epochs of rewarding were completed at Epoch 107. Hence reset PCAP of B and D to 98W.

Fig. 3. KarmaPM power scheduling policy for improving throughput and fairness

Distributing unused power to busy applications Each of these four sockets is initially assigned the user-set power cap of $\mathrm{PCAP}_{\mathrm{Default}}$=98W, which is 65% of the Thermal Design Power (TDP), resulting in a total system-level $\mathrm{PCAP}_{\mathrm{Default}}$=392W (Figure 3(a)). The KarmaPM policy begins at Epoch 20, as illustrated in Figure 3(a). The initial two seconds, corresponding to the first twenty Epochs, were excluded as a warmup duration (refer to Section 5.1). At Epoch 20, the KarmaPM daemon identified that $\mathrm{Bench}_{\mathrm{A}}$ and $\mathrm{Bench}_{\mathrm{B}}$ were experiencing slack, while $\mathrm{Bench}_{\mathrm{C}}$ and $\mathrm{Bench}_{\mathrm{D}}$ were operating under full load. As a result, power usage ($\mathrm{P}_{\mathrm{Usage}}$) for $\mathrm{Socket}_{\mathrm{A}}$ and $\mathrm{Socket}_{\mathrm{B}}$ reduced to 78W and 68W, respectively. Hence, KarmaPM reduces the PCAP at $\mathrm{Socket}_{\mathrm{A}}$ and $\mathrm{Socket}_{\mathrm{B}}$ in Epoch 21 to their current power consumption levels to avoid wastage of power while simultaneously increasing the PCAP of both $\mathrm{Socket}_{\mathrm{C}}$ and $\mathrm{Socket}_{\mathrm{D}}$ by 25W (Figure 3(b)). This situation persisted for the following twenty Epochs. After this period, the KarmaPM daemon noted that $\mathrm{Bench}_{\mathrm{A}}$ had returned to full CPU utilization, while $\mathrm{Bench}_{\mathrm{B}}$ was still displaying slack (Figure 3(c)). KarmaPM then equally distributed the combined surplus power of 30W from $\mathrm{Socket}_{\mathrm{C}}$ and $\mathrm{Socket}_{\mathrm{D}}$ among all busy sockets, resulting in a PCAP of 108W for $\mathrm{Socket}_{\mathrm{A}}$, $\mathrm{Socket}_{\mathrm{C}}$, and $\mathrm{Socket}_{\mathrm{D}}$ (Figure 3(d)).

Rewarding donor applications with surplus power A receiver application is required to reward the donor application once the donor enters a busy state to ensure fairness in the system. KarmaPM computes the average total power donated to a application using the equation shown in Figure 2. The reward power allocated to the donor application is half the average power received by that application and is provided for the same number of total Epochs it donated. Since $\mathrm{Bench}_{\mathrm{C}}$ and $\mathrm{Bench}_{\mathrm{D}}$ each received 10W from $\mathrm{Bench}_{\mathrm{A}}$ over a span of twenty Epochs (from Epoch 22 to 41), KarmaPM subsequently rewarded $\mathrm{Bench}_{\mathrm{A}}$ by granting 5W of power from both $\mathrm{Socket}_{\mathrm{C}}$ and $\mathrm{Socket}_{\mathrm{D}}$ (Figure 3(e)) during the following twenty Epochs (Figure 3(f)). By Epoch 65, KarmaPM recognized that $\mathrm{Bench}_{\mathrm{B}}$ had returned to full CPU utilization (Figure 3(f)). With all four applications now busy, KarmaPM will first reset the PCAP at each socket to $\mathrm{PCAP}_{\mathrm{Default}}$ (Figure 3(g)) before calculating the reward power to $\mathrm{Bench}_{\mathrm{B}}$ from $\mathrm{Bench}_{\mathrm{A}}$, $\mathrm{Bench}_{\mathrm{C}}$, and $\mathrm{Bench}_{\mathrm{D}}$ (see Figure 2). The average power donated by $\mathrm{Socket}_{\mathrm{B}}$ to $\mathrm{Socket}_{\mathrm{A}}$ was 10W for 20 Epochs, while it donated 12.5W each to $\mathrm{Socket}_{\mathrm{C}}$ and $\mathrm{Socket}_{\mathrm{D}}$ for 40 Epochs. Consequently, $\mathrm{Socket}_{\mathrm{A}}$, $\mathrm{Socket}_{\mathrm{C}}$, and $\mathrm{Socket}_{\mathrm{D}}$ will reward $\mathrm{Socket}_{\mathrm{B}}$ by providing 5W, 6.25W, and 6.25W of power, respectively, for 20 Epochs (Figure 3(h)). By Epoch 88, the PCAP at $\mathrm{Socket}_{\mathrm{A}}$ is reset to the user-defined default PCAP, while $\mathrm{Socket}_{\mathrm{C}}$ and $\mathrm{Socket}_{\mathrm{D}}$ continue to reward $\mathrm{Socket}_{\mathrm{B}}$ by contributing 6.25W each for the next 20 Epochs (Figure 3(i)). However, after 10 Epochs, the application running at $\mathrm{Socket}_{\mathrm{C}}$ terminated. Consequently, the PCAP at $\mathrm{Socket}_{\mathrm{C}}$ will be revert to the default user-set PCAP, and $\mathrm{Socket}_{\mathrm{B}}$ will now continue receiving the reward power only from $\mathrm{Socket}_{\mathrm{D}}$ (Figure 3(j)). The PCAP will also be reset at $\mathrm{Socket}_{\mathrm{A}}$ to the default PCAP at Epoch 105 as its application terminated (Figure 3(k)). Finally, $\mathrm{Socket}_{\mathrm{B}}$ would

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

Fig. 4. Speedup from using a Power Manager (PM) with a mix of n applications

continue receiving reward power until Epoch 107, after which its PCAP will revert to the default user-set PCAP (Figure 3(l)).

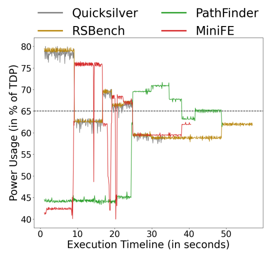
6 Experimental Evaluation

Mixes	Mix1 (OpenMP/Kokkos)	Mix2 (OpenMP/Kokkos)	Mix3 (OpenMP-only)	Mix4 (OpenMP-only)	Mix5 (OpenMP-only)	Mix6
App0	PathFinder	CG	SimpleMOC	CG	CG	SimpleMOC (MPI+OpenMP)
App1	MiniFE (Kokkos)	MiniFE (Kokkos)	PathFinder	SimpleMOC	SimpleMOC	Pennant (MPI-only)
App2	RSBench	CoMD	Pennnant	CoMD	PathFinder	-
App3	Quicksilver	Pennant	Quicksilver	Quicksilver	RSBench	-

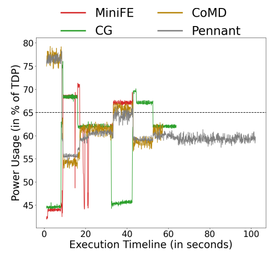
Table 1. Details of the application mixes used for evaluations

This section presents the experimental evaluation of KarmaPM using the co-running mixes shown in Table 1. We created six mixes of co-running applications, as shown in Table 1. The rationale behind these mixes was to create two and four co-running pairs, combine applications that exhibit slack with those that fully utilize CPU resources, and include different parallel programming models (shown in the same Table). The sockets were divided equally among each co-runner in each mix (using the *taskset* command). None of the applications in any mix shared socket-local resources with others in that mix. The SimpleMOC in Mix6 was executed with two MPI ranks, each allocated to a separate socket, and each rank used 18 OpenMP threads with affinity set to the local socket. Pennant in the same Mix6 was executed with 36 MPI ranks spanning across two sockets. We executed each mix ten times and reported the mean value along with a 95% confidence interval. We developed a variant of KarmaPM called SimplePM, which operates similarly to KarmaPM but does not reward the power donor sockets. The work most closely related to SimplePM is PShifter [5], which functions at the cluster level. We ported a single server-level implementation of PShifter to facilitate power transfer among sockets. PShifter differs from SimplePM by its power donation strategy, as it distributes surplus power equally among all four sockets. In contrast, SimplePM improves this approach by allocating surplus power exclusively to the sockets, fully utilizing CPU resources. While we evaluated KarmaPM on an Intel Cooperlake processor, it can be adapted for other Intel and AMD processors by updating the specific MSRs accordingly.

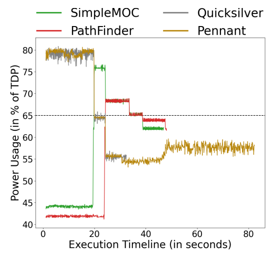
Figure 5 shows the timeline of power scheduling from KarmaPM for each mix at 65% PCAP. We can observe that the reduction in power at one socket increases the power of the other socket without overshooting the system power budget. A similar trend holds at the other two PCAPs. We used the formula shown in Figure 4 to calculate the improvement in throughput over the `Default` execution at each of the three PCAPs by using PShifter, SimplePM, or KarmaPM



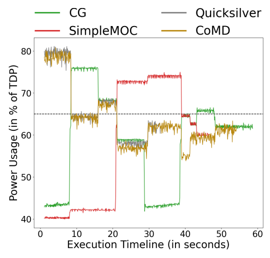
(a) Mix1



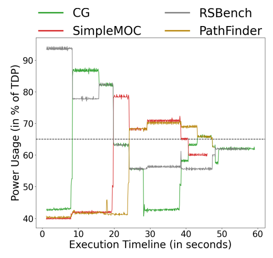
(b) Mix2



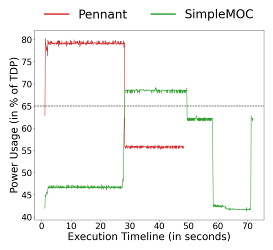
(c) Mix3



(d) Mix4

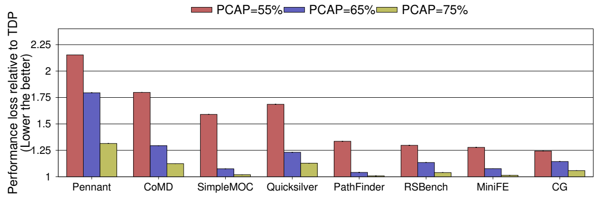


(e) Mix5

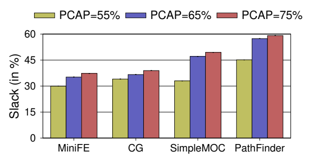


(f) Mix6

Fig. 5. Timeline of power distribution from KarmaPM across sockets at PCAP=65%



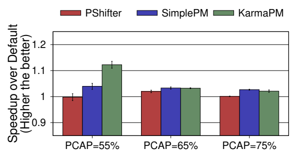
(a) Power sensitivity presented as a loss in performance relative to execution at TDP



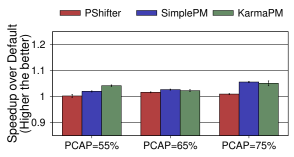
(b) Total slack as a percentage of total execution

Fig. 6. Analysis of power sensitivity and slack at each PCAP

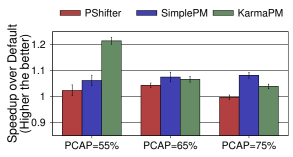
while executing each of our six mixes [12]. The `Default` run uses the same settings as the other three Power Managers (PMs) but does not perform power transfers across sockets. The result of this experiment is shown in Figure 7. Table 2 compares the speedup for individual applications between SimplePM and KarmaPM at each PCAP. The throughput achieved by any of the three power managers (PMs) is influenced by several factors: Point-1) the user-set PCAP and the power sensitivity of each application at that PCAP, Point-2) the percentage of total slack in power donor applications, Point-3) the number of applications in the mix that possess slack, and Point-4) the number of power receiver applications. Figure 6(a) illustrates each application's power sensitivity as performance loss at various PCAPs relative to the execution time at TDP. Notably, at higher PCAPs (65% and 75%), the performance loss is minimal for CG, MiniFE, RSBench, PathFinder, and SimpleMoC. In contrast, Quicksilver, CoMD, and Pennant demonstrate relatively significant performance losses at



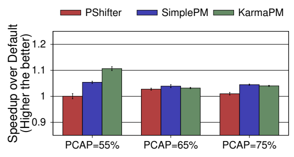
(a) Mix1



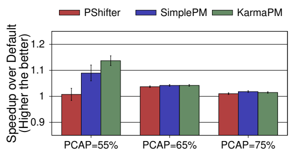
(b) Mix2



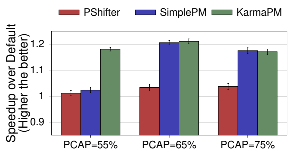
(c) Mix3



(d) Mix4



(e) Mix5



(f) Mix6

Fig. 7. Improvement in overall system throughput at each PCAP

these two PCAPs. Figure 6(b) shows the percentage of total execution time during which slack occurred at each PCAP level for low power sensitivity applications (MiniFE and CG) and high power sensitivity applications (SimpleMoC and PathFinder). During slack periods, each application's power consumption remained below the selected PCAPs, resulting in consistent slack durations for each application across PCAP levels. However, non-slack execution time varies based on an application's power sensitivity, decreasing as the PCAP increases. As a result, the slack percentage increases with higher PCAP levels in SimpleMoC and PathFinder (Figure 6(b)). This rising slack proportion causes these highly power-sensitive applications to appear relatively less sensitive in Figure 6(a).

Performance improvement relative to `Default` (in %)													
Mixes	**Policy**	**PCAP=55%**				**PCAP=65%**				**PCAP=75%**			
		App0	App1	App2	App3	App0	App1	App2	App3	App0	App1	App2	App3
Mix1	PShifter	-0.7	-0.7	0.6	-0.1	0.2	1.2	3.0	3.6	-0.1	0.2	0.3	0.0
	SimplePM	-0.5	3.0	5.4	8.2	0.2	2.0	4.0	7.2	-0.3	0.0	1.3	10.2
	KarmaPM	36.9	8.5	3.8	3.1	3.8	3.1	2.1	4.1	0.5	-0.4	0.1	8.7
Mix2	PShifter	-0.3	0.4	1.2	-0.2	0.8	0.3	3.2	2.2	3.1	1.7	0.5	-0.7
	SimplePM	0.0	0.0	5.5	2.5	0.6	0.5	4.4	5.5	1.9	0.9	5.7	14.5
	KarmaPM	0.5	8.3	4.3	3.8	2.2	1.6	0.1	5.5	0.6	1.4	2.1	16.6
Mix3	PShifter	0.3	7.8	1.5	-0.2	1.2	-0.1	7.6	9.2	1.1	-0.1	-0.8	0.2
	SimplePM	4.7	2.1	3.0	15.2	0.6	-0.1	16.2	14.2	-0.7	-0.3	25.7	10.4
	KarmaPM	43.4	22.6	4.7	17.9	3.9	2.4	11.7	8.9	1.4	0.7	3.4	10.6
Mix4	PShifter	0.4	0.6	1.0	-1.9	-0.5	1.0	4.7	5.9	3.3	1.2	-0.1	-0.5
	SimplePM	0.6	1.1	6.6	13.7	0.2	3.0	6.3	6.2	1.0	0.8	7.4	9.1
	KarmaPM	1.1	32.7	4.0	7.4	1.7	6.7	0.5	3.9	1.7	2.1	5.1	7.3
Mix5	PShifter	0.5	0.3	-2.5	4.8	2.3	3.5	1.0	8.2	2.7	1.3	-0.1	0.2
	SimplePM	2.0	24.7	-0.4	10.9	3.7	2.8	1.0	9.4	1.9	2.7	0.1	2.6
	KarmaPM	2.3	37.3	9.7	8.7	3.0	6.0	3.7	4.1	2.2	2.7	0.6	0.5
Mix6	PShifter	2.7	0.6	-	-	5.4	1.0	-	-	-0.3	8.1	-	-
	SimplePM	1.0	3.1	-	-	-0.8	53.1	-	-	-0.7	49.2	-	-
	KarmaPM	26.3	15.2	-	-	1.6	48.1	-	-	0.6	45.8	-	-

Table 2. Speedup by three PMs for individual applications across mixes at each PCAPs

6.1 Speedup at Higher PCAPs (65% and 75%)

Mix6 achieved significantly higher throughput at higher PCAPs with SimplePM and KarmaPM compared to all other mixes (53% at PCAP=65% and 49% at

PCAP=75%). In this case, a single power-sensitive application (Pennant) received the donated power, leading to substantial performance gains. As shown in Figure 5(f), SimpleMOC donated power from its slack region to Pennant and briefly received reward power in return. However, since Pennant terminated shortly thereafter, the throughput remained similar for both SimplePM and KarmaPM. Despite overall system throughput being the same in this mix, unlike SimplePM, KarmaPM promoted fairness by improving the speedup of SimpleMOC (see Table 2). Mix3 showed marginally better throughput with SimplePM (7.5% at PCAP=65% and 8.2% at PCAP=75%) and KarmaPM (6.6% at PCAP=65% and 3.9% at PCAP=75%). It was primarily due to the overlapping slack regions in SimpleMoC and PathFinder. However, at 75% PCAP, SimplePM achieved comparatively higher throughput than KarmaPM. This difference arises from KarmaPM's reward policy, prioritising fairness by distributing speedup improvements across all applications. As shown in Table 2, SimplePM only enhanced the speedup of Pennant and Quicksilver in Mix3, whereas KarmaPM promoted fairness by boosting the speedup of other co-runners. Mix2 and Mix4 contain two applications with slack regions (CG and MiniFE in Mix2, CG and SimpleMoC in Mix4). However, unlike Mix3, their slack periods do not fully overlap, leading to only minor and similar throughput improvements at both PCAPs (up to 5.6%). Although CG experiences slack twice, only the first instance overlaps with another application's slack. Mix1 has a single application sensitive to these two PCAPs (Quicksilver), whereas Mix5 lacks any of the three power-sensitive applications, so these two pairs performed similarly to `Default`. Overall, at each of these two high PCAPs, KarmaPM attained fairness in each mix by enhancing the speedup of each application through its power reward policy (Table 2). Overall, PShifter performed worse than both SimplePM and KarmaPM across all mixes and PCAPs, including PCAP=55%. It was designed for cluster environments with many nodes, where a node's surplus power is evenly distributed among the rest of the nodes in the cluster. In a large cluster, the proportion of nodes experiencing slack (at the same time) is lower than that of sockets experiencing slack in a single multiprocessor server. While effective at the cluster level, this approach is less suitable for a single-server environment.

6.2 Speedup at Lower PCAP (55%)

At lower PCAP, KarmaPM performed better than SimplePM in all mixes. The improvement in throughput in KarmaPM relative to SimplePM were 8% in Mix1, 2.2% in Mix2, 14.3% in Mix3, 5% in Mix4, 4.4% in Mix5, and 15.4% in Mix6 (geomean improvement of 6.6% over SimplePM and 12% over `Default`). At the same time, owing to its novel reward policy, unlike SimplePM, it significantly improved the speedup of the slack-experiencing applications in each mix (Table 2). MiniFE and CG were exceptions. Despite experiencing slack, their insensitivity to power means that even at low PCAP (Figure 6(a)), receiving reward power results in only marginal performance gains. In contrast, SimpleMoC and PathFinder, despite experiencing slack, showed notable performance improvements when receiving reward power. Among the four applications that do not experience slack, QuickSilver, CoMD, and Pennant exhibited high sensitivity to power transfers at low PCAPs. As a result, their performance was significantly influenced by both receiving and donating power. KarmaPM achieved the highest throughput improvements in Mix3 and Mix6. It was due to the high-power sensitivity applications in both mixes and overlapping slacks in Mix3 (SimpleMOC and PathFinder). The effect of KarmaPM was least visible in Mix2 because MiniFE had a comparatively short slack region (Figure 6(b)), with some slack instances not overlapping with that in CG (Figure 1(a)). SimplePM's performance was best in Mix5 due to three applications exhibiting slack (CG, SimpleMoC, and PathFinder).

At present, KarmaPM redistributes unused power from one application to others in equal shares without considering the power sensitivity of the receiver applications. In future work, we intend to improve KarmaPM by making power distribution sensitivity-aware. Similarly, KarmaPM will transfer reward power based on the power sensitivity of the recipient application.

6.3 Energy Usage

Figure 8 shows the geometric mean energy consumption of SimplePM and KarmaPM at each PCAP for all mixes relative to `Default`. Overall, KarmaPM consumed less energy than `Default` across all mixes, with savings ranging from -0.1% to 7.9%. Except for Mix3, KarmaPM's energy use was comparable to or lower than that of SimplePM. The higher energy usage in Mix3 arises from SimplePM achieving greater system throughput than KarmaPM at higher PCAPs (see Section 6.1). In Mix2, Mix4, and Mix5, both KarmaPM and SimplePM showed similar energy usage due to CG, a low power-sensitive application, resulting in comparable performance at higher PCAPs. In Mix1 and Mix6, although both KarmaPM and SimplePM achieved similar throughput, KarmaPM consumed 4.9% and 5.3% less energy, respectively, due to the presence of two highly power-sensitive applications that experienced slack and benefited from KarmaPM's reward mechanism.

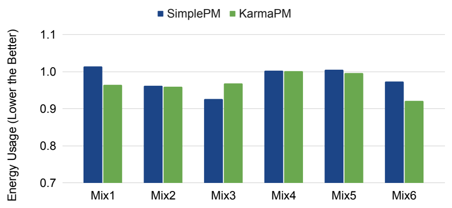


Fig. 8. Geomean energy usage at all PCAPs relative to `Default` for each mix

7 Conclusion

This paper proposes a programming model oblivious power management solution for multiprocessor servers to improve system throughput and application fairness under a limited power budget for co-running parallel applications. Our approach uses a novel reward-based scheme to boost the performance of a power donor application by rewarding them with additional power usage based on their previous power donation activities to other co-running applications. Our empirical results demonstrate that our reward-driven power management solution can achieve better throughput and energy savings than traditional approaches.

8 Acknowledgement and Artifact Availability

This research was supported by the Google PhD Fellowship 2022 program. The artifact of this paper is available in the Zenodo repository [8].

References

1. TOP500 (November 2024), `https://top500.org/lists/top500/2024/11/`
2. ECP proxy applications (Released), `https://proxyapps.exascaleproject.org`
3. Costero, L., Igual, F.D., Olcoz, K.: Dynamic power budget redistribution under a power cap on multi-application environments. SUSCOM (2023). https://doi.org/10.1016/j.suscom.2023.100865
4. Ding, J., Hoffmann, H.: DPS: Adaptive power management for overprovisioned systems. In: SC (2023). https://doi.org/10.1145/3581784.3607091
5. Gholkar, N., Mueller, F., Rountree, B., Marathe, A.: PShifter: Feedback-based dynamic power shifting within hpc jobs for performance. In: HPDC (2018). https://doi.org/10.1145/3208040.3208047
6. Huang, D., Costero, L., Atienza, D.: Is the powersave governor really saving power? In: CCGrid (2024). https://doi.org/10.1109/CCGrid59990.2024.00039
7. Inadomi, Y., Patki, T., Inoue, K., Aoyagi, M., Rountree, B., Schulz, M., Lowenthal, D., Wada, Y., Fukazawa, K., Ueda, M., et al.: Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing. In: SC (2015). https://doi.org/10.1145/2807591.2807638
8. Kumar, S., Kumar, V.: KarmaPM: Reward-Driven Power Manager (2025). https://doi.org/10.5281/zenodo.15602982
9. Lee, S., Lowenthal, D.K., De Supinski, B.R., Islam, T., Mohror, K., Rountree, B., Schulz, M.: I/o aware power shifting. In: IPDPS (2016). https://doi.org/10.1109/IPDPS.2016.15
10. Patel, T., Tiwari, D.: PERQ: Fair and efficient power management of power-constrained large-scale computing systems. In: HPDC (2019). https://doi.org/10.1145/3307681.3326607
11. Patki, T., Lowenthal, D.K., Rountree, B., Schulz, M., de Supinski, B.R.: Exploring hardware overprovisioning in power-constrained, high performance computing. In: ICS (2013). https://doi.org/10.1145/2464996.2465009
12. Roy, R.B., Patel, T., Tiwari, D.: Satori: Efficient and fair resource partitioning by sacrificing short-term benefits for long-term gains. In: ISCA (2021). https://doi.org/10.1109/ISCA52012.2021.00031
13. Solórzano, A.L.V., Sato, K., Yamamoto, K., Shoji, F., Brandt, J.M., Schwaller, B., Walton, S.P., Green, J., Tiwari, D.: Toward sustainable hpc: In-production deployment of incentive-based power efficiency mechanism on the fugaku supercomputer. In: SC (2024). https://doi.org/10.1109/SC41406.2024.00030
14. Wang, B., Miller, J., Terboven, C., Müller, M.: Operation-aware power capping. In: Euro-Par (2020). https://doi.org/10.1007/978-3-030-57675-2_5
15. Zhang, H., Hoffmann, H.: Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques. In: ASPLOS (2016). https://doi.org/https://doi.org/10.1145/2872362.2872375
16. Zhang, H., Hoffmann, H.: PoDD: Power-capping dependent distributed applications. In: SC (2019). https://doi.org/10.1145/3295500.3356174