Cooperative Slack Management: Saving Energy of Multicore Processors by Trading Performance Slack between QoS-Constrained Applications

Cooperative Slack Management: Saving Energy of Multicore Processors by Trading Performance Slack Between QoS-Constrained Applications

MEHRZAD NEJAT, MADHAVAN MANIVANNAN, MIQUEL PERICÀS, and PER STENSTRÖM, Chalmers University of Technology, Sweden

Processor resources can be adapted at runtime according to the dynamic behavior of applications to reduce the energy consumption of multicore processors without affecting the Quality-of-Service (QoS). To achieve this, an online resource management scheme is needed to control processor configurations such as cache partitioning, dynamic voltage-frequency scaling, and dynamic adaptation of core resources.

Prior State-of-the-art has shown the potential for reducing energy without any performance degradation by coordinating the control of different resources. However, in this article, we show that by allowing short-term variations in processing speed (e.g., instructions per second rate), in a controlled fashion, we can enable substantial improvements in energy savings while maintaining QoS. We keep track of such variations in the form of performance slack. Slack can be generated, at some energy cost, by processing faster than the performance target. On the other hand, it can be utilized to save energy by allowing a temporary relaxation in the performance target. Based on this insight, we present Cooperative Slack Management (CSM). During runtime, CSM finds opportunities to generate slack at low energy cost by estimating the performance and energy for different resource configurations using analytical models. This slack is used later when it enables larger energy savings. CSM performs such trade-offs across multiple applications, which means that the slack collected for one application can be used to reduce the energy consumption of another. This cooperative approach significantly increases the opportunities to reduce system energy compared with independent slack management for each application. For example, we show that CSM can potentially save up to 41% of system energy (on average, 25%) in a scenario in which both prior art and an extended version with local slack management for each core are ineffective.

CCS Concepts: • **Computer systems organization** → **Multicore architectures**; **Reconfigurable computing**; • **Hardware** → *Enterprise level and data centers power issues*; **Chip-level power issues;**

Additional Key Words and Phrases: Multicore processors, QoS, DVFS, cache partitioning, dynamic core resizing, performance and energy modeling

ACM Reference format:
Mehrzad Nejat, Madhavan Manivannan, Miquel Pericàs, and Per Stenström. 2022. Cooperative Slack Management: Saving Energy of Multicore Processors by Trading Performance Slack Between QoS-Constrained Applications. *ACM Trans. Arch. Code Optim.* 19, 2, Article 21 (January 2022), 27 pages.
https://doi.org/10.1145/3505559

This research has been funded by the European Processor Initiative under the project number 800928.
Authors' address: M. Nejat, M. Manivannan, M. Pericàs, and P. Stenström, Chalmers University of Technology, Department of Computer Science and Engineering SE-412 96 Göteborg Sweden; emails: {nejatm, madhavan, miquelp, per.stenstrom}@chalmers.se.

1544-3566/2022/01-ART21 $15.00
https://doi.org/10.1145/3505559

1 INTRODUCTION

Running applications at maximum processing speed, beyond the applications' Quality-of-Service (QoS) requirements, can waste substantial amounts of energy [30, 31]. Instead, by associating a performance target with each application based on its QoS requirements —expressed in, say, instructions per seconds (IPS) —one can build resource management frameworks that dynamically adapt processor resources to save energy while meeting performance targets.

There are prior resource management frameworks proposed for improving the efficiency of using hardware resources in multicore platforms while respecting performance constraints. In one line of research, a mix of latency-critical and best-effort applications is considered in a data-center workload [10, 28, 35]. Here, a common goal is to improve throughput and resource utilization by reclaiming the excess resources for best-effort applications after meeting the performance targets of latency-critical applications. In another line of research [30–33], the goal is to reduce energy/power consumption while meeting individual performance targets of all applications. One approach [33] uses core mapping and dynamic voltage-frequency scaling (DVFS) as the control knobs. Another approach uses coordinated management of the Last Level Cache (LLC) partitioning together with per-core DVFS [30, 32] extended with dynamic adaptation of the core microarchitecture [31]. In this work, we assume that every application in the workload is associated with a performance target according to a baseline allocation of processor resources.

A key insight driving this study is that allowing applications to deviate temporarily from a fixed performance target can enable additional energy savings while ensuring that the performance targets are satisfied over a longer time interval, referred to as a *QoS window*. Within that window, when the application runs faster compared with the baseline performance target, we say that it *generates* slack and when it runs slower, we say that it *consumes* slack. The goal is to find opportunities to generate slack at a lower energy cost compared with the energy savings achieved by consuming it. For example, when the LLC share of a memory-intensive application is increased, it may enjoy a performance boost that generates slack at no energy cost. This slack can be consumed later, within the same QoS window, to save energy, for example, by reducing the core voltage-frequency (VF). It can also be consumed by a redistribution of the allocated LLC share to other applications. In this way, slack can be *transferred* from one application to another, which enables more trade-offs to save processor energy. Meanwhile, by keeping track of the slack balance on each core, the resource manager can ensure that the performance of every application continuously remains ahead of the baseline targets.

This article proposes Cooperative Slack Management (CSM), an online resource management scheme that leverages the insight regarding slack to improve processor energy efficiency. It continuously monitors dynamic program characteristics such as the cache-miss versus cache-size profile. Using simple analytical models, it estimates the effect of different resource configurations on performance and energy at runtime. These configurations include different partitionings of the LLC as well as the size and VF of each core. This way, it attempts to find the right opportunities and amounts for generating/consuming slack across all applications. It can save substantial energy by making trade-offs across the processor configuration *space* at different points in *time* while respecting the QoS of all applications.

There are two main challenges in implementing this approach. First, the overheads must be small enough to enable resource management at regular intervals during runtime. Second, assuming no prior knowledge about the applications, CSM uses statistics collected from hardware performance monitoring counters (PMCs) and analytical models to explore the processor configuration space at each invocation. Therefore, it is prone to modeling error that can affect both the energy savings and QoS. To address the first challenge, we used simplified models together with a heuristic

algorithm that reduces the dimensions of the problem at several levels while performing a cost-benefit analysis on slack across different cores. To address the second challenge, we propose a dynamic sampling technique that gradually improves the modeling accuracy over runtime. In a hybrid approach, it can actively enforce interesting configurations to be sampled as soon as possible as well as passively collect samples based on the decisions of CSM.

The main contributions of this article can be summarized as follows.

(1) We make two important insights that motivate this study: (a) the possibility to generate short-term performance slack at a lower energy cost compared with the energy-saving enabled by consuming it; and (b) by transferring slack from one application to another through the shared resource, it is possible to further improve energy efficiency.
(2) We present Cooperative Slack Management (CSM), a low-overhead online resource management scheme to improve processor energy efficiency without affecting the QoS of any application in the workload through slack management.
(3) We design a dynamic sampling technique to improve modeling accuracy during runtime with a hybrid approach that supports passive and active sampling of selected configurations.
(4) We implement a local slack management algorithm that runs independently on each core to provide a quantitative comparison between cooperative versus local slack management in different scenarios.

We experimentally evaluate several different scenarios regarding the workloads, modeling assumptions, performance targets, number of cores, and both adaptive and fixed core architectures. Furthermore, we analyze the sensitivity of CSM energy savings to different parameters. We show that CSM can save substantially more energy compared with prior state-of-the-art [31] and when it is extended with local slack management. The most significant improvements are achieved in scenarios in which the performance target is high, the core architecture is fixed, and when there is high contention in LLC, that is, all applications are sensitive to their baseline LLC allocation. For example, we show that CSM can potentially save up to 41% and, on average, 25% energy in a scenario in which the energy-saving with the prior state-of-the-art and its extended version is below 4% and, on average, 0.4% and 1.5%, respectively.

The rest of this article is organized as follows. Section 2 explains the background and motivation for this work. The proposed framework is described in Section 3. Sections 4 and 5 present the experimental methodology and results, respectively. The related work is discussed in Section 6. We present our conclusions in Section 7.

2 BACKGROUND AND MOTIVATION

This section establishes basic assumptions and definitions about the resource management framework in this study. It also provides insights that motivate the proposed approach described in Section 3.

2.1 Baseline System

In this work, we study a hardware resource management scheme in the context of a multicore processor that consists of N cores with a private L1 and L2 cache attached to each core and a shared LLC. The LLC ways are dynamically partitioned between cores without overlap, that is, each way is allocated to only one core. The frequency of each core is controlled independently using per-core DVFS. We also study a simple adaptive core architecture such that the resource manager (RM) can select between a few core sizes by deactivating sections of core components, including issue width, reorder buffer, reservation stations, and load/store queues. More details on these resources are presented in Section 3.2. In the rest of this article, a resource configuration

C for a particular application refers to the combination of LLC allocation together with core size and VF.

We assume that a multi-programmed workload consisting of N independent applications is scheduled on the N-core processor without any process migration. We also assume that the baseline resource configuration (C_{base}) satisfies the QoS requirements of all applications. The RM may change the configurations during runtime as long as the performance delivered to each application remains greater than or equal to the performance achievable with C_{base}. However, short-term variations in processing speed are acceptable if the performance target is met at user-defined intervals, called *QoS windows*. More specifically, for each QoS window that includes a certain number of instructions, the execution time must be less than or equal to that with C_{base}. Inside each QoS window, the RM is invoked regularly after executing a fixed number of instructions, which we refer to as *RM intervals*.

2.2 Slack Management Potential

At each RM interval, the RM may find a set of configurations that can improve energy efficiency. However, it needs a way to keep track of variations in processing speed compared with C_{base}. Therefore, we define a parameter called *slack* as

$$s = (TPI(C_{base}) - TPI(C)) \times IC_{RM}. \quad (1)$$

$TPI(C)$ refers to the average time per instruction with configuration C and IC_{RM} denotes the instruction count of the RM interval. For example, 1 ms of slack means that this RM interval is executed 1 ms faster compared with using C_{base}. By calculating s, the RM knows exactly how much reduction in processing speed is acceptable in a future interval without missing the performance target. Therefore, we accumulate the slack values (both positive and negative) over the past RM intervals into a parameter called *Slack Deposit* (SD). Based on this, the performance target is met as long as SD remains greater than or equal to zero by the end of the QoS window. This form of performance target enables new trade-offs to save energy, as explained in the rest of the section.

If sufficient positive SD has been collected over the past RM intervals, the processing speed of the current interval can be safely reduced below the baseline (C_{base}) without affecting QoS. In other words, the application progress has been so much faster compared with C_{base} that there is enough time to relax the processing speed without lowering the performance below C_{base}. However, generating that slack in the first place can impose a certain energy cost. Therefore, to achieve a net energy saving, we need to find opportunities in which the cost of generating slack (running faster) is low and the benefit of consuming slack (relaxing) is high.

To demonstrate the potential of this idea, we analyze an example application. Figure 1(a) shows the simulation results for a selected region of bzip2 from *SPEC CPU2006* benchmarks, executed for 100-M instructions with a fixed core size and different frequencies and LLC allocations. The X-axis shows the execution time normalized to that with C_{base}, which is a mid-range frequency level (VF5) and an allocation of 2-MiB of LLC in this example. The Y-axis shows the normalized energy (to C_{base}) that includes the core, the cache hierarchy, and memory accesses. A detailed description of the experimental methodology is presented in Section 4. Each curve in this figure represents the DVFS range for a particular LLC allocation.

If the performance constraint is strictly set to C_{base}, all configurations with a normalized execution time greater than one are not allowed. This is the case when there is no sufficient SD and the processing speed cannot be relaxed below the baseline. Therefore, the minimum energy points highlighted by red circles can be selected. We observe that some of these points are slightly faster than C_{base}, creating a small amount of slack. This is because of the limited range and granularity of DVFS. However, it is possible to generate more slack at some energy cost, as depicted by the

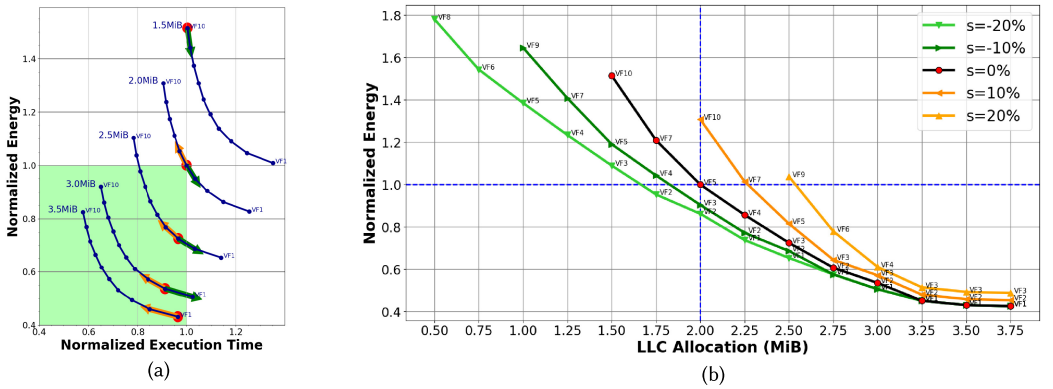


Fig. 1. Normalized energy versus execution time of bzip2 at different resource configurations (a) and minimum energy points as a function of LLC allocation for different slack targets (b).

orange arrows. A key observation is that the energy cost to slack ratio, that is, the slope of the orange arrows, reduces with a larger LLC. In other words, it becomes cheaper to buy slack given more cache. On the other hand, given a large enough SD, we can relax the performance constraint and save more energy. This is depicted by the green arrows. In this case, the energy benefit to slack ratio, that is, the slope of the green arrows, increases with a smaller LLC. In other words, using slack is more beneficial at lower cache allocations. This example shows the possibility of achieving a net energy saving by generating and consuming slack at the right conditions, which is strongly dependent on the LLC partitioning at hand.

To further elaborate on the relation between LLC partitioning and slack management, we analyze the same example from a different perspective in Figure 1(b). Here, the X-axis represents the allocation of LLC to this application while the Y-axis is the normalized energy, similar to Figure 1(a). Again, we start with the initial performance constraint that results in the minimum energy points discussed earlier (red circles in Figure 1(a)). These points are depicted on the black curve in Figure 1(b), which shows how the energy changes with the LLC partition size, and the limited range of acceptable allocations with the performance constraint. To demonstrate the effect of slack generation or consumption, we simply shift the constraint to shorter or longer execution times. For example, by shifting the constraint to 90% of C_{base} execution time, we can generate at least 10% slack. The orange curves in Figure 1(b) show how such slack generation targets change the minimum energy points. On the other hand, we can relax the constraint, for example, to 110% of C_{base} execution time, and allow a negative slack value of −10%. The green curves in the figure show the effect of such slack consumption scenarios. A key observation in this figure is that we can find "*energy equivalent*" points that lie approximately on the same horizontal line. This has two implications: (1) Given sufficient slack, this application can give up a part of its LLC share to others at no energy cost. (2) Given a larger LLC share, this application can generate more slack at no energy cost. These two observations indicate the possibility of transferring slack from one application to another, enabling more energy-saving opportunities.

To summarize, we make the following important insights. Slack can be generated on each core when the energy cost is low and consumed later at an opportune time to earn a net energy saving. The cost and benefit of slack generation and consumption vary depending on dynamic application characteristics and resource configurations at different points in time. Furthermore, slack can be transferred from one application to another to find better opportunities to reduce system energy. As long as SD on each core remains above a lower bound, the performance targets of all

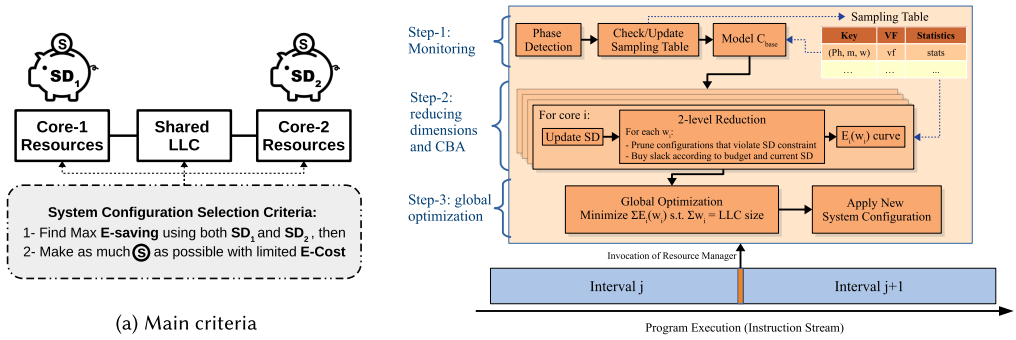


Fig. 2. Overview of the proposed resource management scheme.

applications are satisfied. To exploit these properties of slack for saving processor energy, we present a CSM scheme in the next section.

3 CSM: COOPERATIVE SLACK MANAGEMENT SCHEME

This section introduces **CSM**, an online multicore resource management scheme. Its goal is to improve energy efficiency through making trade-offs between slack generation and consumption across applications. This section first offers an overview of CSM in Section 3.1 before going through the design in detail in Sections 3.2 through 3.6. Finally, we discuss overheads and scalability of CSM in Sections 3.7 and 3.8, respectively.

3.1 System Overview

Figure 2 shows an overview of the proposed scheme and its main components. The RM can be implemented as a lightweight software handler that is invoked on each core at regular intervals after executing a fixed number of instructions.[1] Its objective is to evaluate a range of possible configurations for every core and their effect on performance and energy before selecting the system configuration for the upcoming interval. Figure 2(a) shows the main criteria for performing this selection in a simplified form for a 2-core system. It first looks for a system configuration that achieves maximum energy saving by using the available SD on each core. Then, starting from this point, it attempts to make as much slack (s) as possible with a limited energy cost. While the first criterion attempts to maximize energy saving during the upcoming interval, the second criterion deviates from this goal to facilitate larger energy savings in the future intervals. This trade-off can be controlled by setting an energy budget for slack generation, which is further discussed in the rest of this section.

This operation requires several components, as illustrated in Figure 2(b). The overall procedure can be summarized as follows. It starts with *Step-1*, which monitors the application progress by evaluating the past interval and collecting the required statistics. During *Step-2*, analytical models based on sampling are used to estimate performance slack and energy for different resource configurations, as explained in Sections 3.5 and 3.6. Using these estimations, the RM prunes the resource configuration space while performing a *cost-benefit analysis (CBA)* to decide how much slack should be generated/consumed for any possible LLC allocation. This step reduces the

[1]This is a design choice that is conveniently modeled by our simulation framework. However, it is also possible to modify the design to operate based on fixed time intervals instead.

dimensions of processor resources by producing an energy curve as a function of LLC allocation for each core, as explained in Section 3.3. Finally, the *Global Optimization* function in *Step-3* finds an optimum LLC partitioning that minimizes the sum of energy values over all cores, as detailed in Section 3.4. The selected resource configurations, based on the optimum LLC partitioning, are applied to the system before returning to the program execution.

3.2 Resource Control Knobs

In this work, we study different configurations of the following processor resources.

DVFS: This is a widely used mechanism for energy/power management in most processors. We assume per-core DVFS, which is studied and implemented, for example, in [5, 16, 19, 22]. Here, we consider 10 VF settings that can be independently selected for each core, as detailed in Section 4.2.

Core Scaling: Adaptive core architectures have been studied in previous work [2, 6, 23, 24, 31] as an effective approach in reducing processor energy consumption. While there are many different adaptive architectures proposed, we consider a simple mechanism with minimal overhead. In this mechanism, deactivation logic is added to several components: issue width, reorder buffer, reservation stations, and load/store queues. Therefore, instead of changing the core architecture, its size is reduced by deactivating a section of these components. To perform this at runtime, the core resizing logic takes the following steps: (1) It halts the fetch of new instructions. (2) It waits until the instructions in the pipeline are committed. (3) It deactivates/reactivates a section of the core components. (4) It restarts the instruction fetch. A similar mechanism has been previously studied in [31]. In this work, we consider two different core sizes, as detailed in Section 4.2. We also study the proposed scheme with a fixed core size. The overheads are analyzed in Section 3.7.

LLC Partitioning: Cache partitioning is an effective tool to avoid interference between applications in the LLC. This is especially important for achieving performance predictability and providing QoS guarantees. There are different mechanisms proposed for partitioning the cache capacity. In this work, we use a mechanism that controls the allocation of cache ways to each core, which is implemented, for example, in Intel [17] and Qualcomm [46] products.

3.3 Cost-Benefit Analysis on Slack

The proposed slack management approach is centered around a heuristic algorithm that decides how much slack should be generated/consumed on each core under different conditions. As established in Section 2, these decisions should be made depending on resource configuration, especially the partitioning of the shared LLC. The proposed heuristic approach breaks down the problem into a local and a global component. The objective of the local component is to evaluate the slack tradeoffs for a range of possible LLC allocations (w_i) for each core i and prune the local configuration space before entering the global optimization that explores different distributions of LLC shares.

As depicted in Figure 2(b), the local component for core i, marked as *Step-2*, starts by updating the SD value based on the statistics collected during the past interval. Next, it estimates a slack (s) value for different core configurations, using Equation (1) and the analytical models explained later. All configurations that violate the following lower-bound on SD are pruned:

$$SD + s \geq \epsilon \tag{2}$$

Here, the value of ϵ can be adjusted according to the criticality of the QoS targets. For example, with $\epsilon = 0$, the value of SD is never allowed to become negative. In other words, the performance must remain ahead of the baseline target at all points in time. However, in the case of a non-critical

ALGORITHM 1: The 2-Level Reduction Pseudo-Code

Inputs:
$P(w) = \{(E, s)\}$: the set of energy-slack points for all core configurations, given LLC allocation w
SD: The current value of slack deposit
ϵ: The lower bound on SD
SD_{thr}: The upper threshold on SD
E_{budget}: The energy budget for slack generation
Output:
$E(w)$: Energy curve as a function of LLC allocation w

1: **function** BEST_POINT(P)
2: $P_{valid} = \{(E, s)$ if $(SD + s \geq \epsilon)$ for $(E, s) \in P\}$
3: Find $(E^*, s^*) \in P_{valid}$ that has min. E
4: **if** $SD > SD_{thr}$ **then**
5: Return (E^*, s^*)
6: **else**
7: Find $(E, s) \in P_{valid}$ with max s, such that $E - E^* \leq E_{budget}$
8: Return (E, s)
9: **end if**
10: **end function**

11: **for** $w \in W_{range}$ **do**
12: (E, s) = Best_Point(P(w))
13: E(w) = E
14: **end for**

ALGORITHM 2: Global Optimization Pseudo-Code

Inputs:
Curves = $\{E_i(w_i), \forall i \in [0, N]\}$: Energy curves as a function of LLC allocation w_i for each core i
Output:
$V^* = \{w_i^*, \forall i \in [0, N]\}$: Optimum LLC Allocation

1: **function** REDUCE($E_x(w_x)$,$E_y(w_y)$)
2: **for** each possible allocation $w_{xy} = w_x + w_y$ **do**
3: $E_{xy}(w_{xy})$ = Minimum($[E_{xy} = E_x + E_y$ for different $(w_x, w_y)]$)
4: Store the allocation $V_{xy}(w_{xy}) = \{w_i, \forall i \in (x \cup y)\}$
5: **end for**
6: **Return** $E_{xy}(w_{xy})$
7: **end function**

8: **while** Length(Curves) > 2 **do**
9: **for** each pair $(k, k + 1)$ in Curves **do**
10: Replace (E_k, E_{k+1}) with **REDUCE**(E_k, E_{k+1})
11: **end for**
12: **end while**
13: $E_x(w_x)$, $E_y(w_y)$: the two remaining curves
14: Find the (w_x^*, w_y^*) that minimizes $E_x + E_y$ such that $w_x^* + w_y^*$ = LLC size
15: **Return** Optimum allocation $V^* = V_x(w_x^*) \cup V_y(w_y^*)$

QoS target, setting a small negative value for ϵ can enable additional energy saving with a small and bounded effect on QoS. This is analyzed quantitatively in Section 5.3.

After the first level of reduction based on the SD constraint, an energy value is estimated for each remaining configuration using the analytical models. This forms a set of energy-slack pairs $\{(E, s)\}$. During the second level of reduction, one member of this set is selected as the best configuration as follows. First, the minimum energy point (E^*, s^*) is found. If the current SD value is already above a predefined upper threshold $(SD > SD_{thr})$, the minimum energy point is selected. Otherwise, it considers buying additional slack at some energy cost compared with E^*. We set a predefined energy budget, for example, 6% of the baseline system energy, for buying slack. These threshold values can be used to adjust and adapt the behavior of CSM. For example, using a larger value for SD_{thr} or energy budget can raise the available SD at future intervals to potentially exploit larger energy-saving opportunities. However, it increases the energy costs to generate slack at each interval. In this study, we set the values for these thresholds empirically based on experimental observations. This is further discussed in Section 5.3. CSM can be extended to adjust these values at runtime according to the workload characteristics. We leave this extension for future work.

The pseudo-code of this process is presented in Algorithm 1. The first level of reduction, based on Equation (2), is performed at line 2; lines 3 to 9 represent the second level of reduction. Consequently, this process creates a one-dimensional energy curve as a function of LLC allocation $(E_i(w_i))$ for each core i, as presented in lines 11 to 14.

The explained process implicitly sets the amount of slack to be generated/consumed in advance for all possible LLC allocations to each core before exploring different partitionings of the LLC. It potentially buys more slack for larger allocations and consumes more slack for smaller allocations. As shown in Section 2, the energy curves for different slack targets converge toward larger LLC allocations (see Figure 1(b)). Therefore, given a certain energy budget, a larger amount of slack can potentially be generated as the allocation increases.

3.4 Global Optimization

Once the energy curves $\{E_i(w_i)\}$ are available for all cores, the last part of the heuristic (*Step-3* in Figure 2(b)) is an optimization algorithm that finds an allocation $V = \{w_i\}$ of LLC partitions that minimizes the sum of energy values over all cores. For this purpose, we use the approach presented in Algorithm 2. The "*Reduce*" function at lines 1 to 7 reduces a pair of energy curves (x and y) into one (xy) by finding the minimum total energy (E_{xy}) for different allocations to that pair (w_{xy}). It also stores the corresponding allocation vector (V_{xy}) as a function of w_{xy}, for all cores that belong to the pair. This function is used recursively at lines 8 to 12 until two curves are left (line 13). The optimum LLC allocation V^* is found at lines 14 to 15 that minimizes the sum of two remaining energy values such that the total allocation equals the LLC size. This process requires $N-1$ reductions for an N core system. A similar optimization approach has been previously used in [14, 30–32].

3.5 Analytical Modeling

The proposed RM algorithm requires an estimation of performance and energy for a range of resource configurations. We assume several hardware PMCs that provide online measurement results for different performance and power components, which are explained in more detail in the rest of this section. Thus, accurate measurements are available for the past RM intervals. However, the RM needs a prediction for the upcoming interval for several configurations that may not have been used before.

Based on a simplified assumption that the program behavior does not change in the upcoming interval, analytical models are used to estimate the performance and energy difference between two resource configurations: (1) a target configuration $C_t = (m_t, f_t, w_t)$, where m, f, and w refer to the core size, frequency, and LLC allocation, respectively; and (2) the configuration used during an earlier interval, which we call a *sampled configuration* $C_s = (m_s, f_s, w_s)$ for which measurement results are available. Hence, the execution time (T) of the upcoming interval using C_t is calculated as:

$$T(C_t) = \left(T_{0,s} \times \frac{D(m_s)}{D(m_t)} + T_{BP,s} + T_{Cache,s} \right) \times \frac{f_s}{f_t} + \frac{M(w_t)}{M(w_s)} \times T_{mem,s} \tag{3}$$

$$T_{0,s} = T_s - T_{BP,s} - T_{cache,s} - T_{mem,s} \tag{4}$$

Here, the execution time is broken down into three main components. First, $T_{0,s}$ corresponds to the cycles spent on processing instructions during the sampled interval with C_s. It is calculated by subtracting the stall time due to branch prediction ($T_{BP,s}$), cache accesses ($T_{cache,s}$), and memory accesses ($T_{mem,s}$) from the total execution time of the sampled interval (T_s), as shown in Equation (4). This component is scaled with the inverse ratio of the instruction dispatch width, denoted by $D(m)$. With a fixed core size, and when the sizes are the same, this ratio turns into one. The second component is the sum of $T_{BP,s}$ and $T_{cache,s}$, which is affected by the core frequency similar to $T_{0,s}$. Hence, these components are scaled with the inverse ratio of the core frequency. The last component is the memory access time ($T_{mem,s}$). We simply scale this component with the number of LLC load misses, denoted by $M(w)$ for LLC allocation w. In order to estimate $M(w)$ for any w, we assume an Auxiliary Tag Directory (ATD), proposed by Qureshi and Patt [38], for each core in the system. The ATD emulates the operation of the main tag directory to detect whether each memory request hits in any recency position, given the full cache size. It includes a hit counter for each recency position as well as a miss counter. The number of misses with w cache ways is calculated as the total number of hits in recency positions greater than w, plus the number of ATD misses. In addition to the ATD, PMCs are needed that count the number of cycles spent on computation, as well as the stall cycles due to branch prediction, cache, and memory accesses.

After estimating the performance for the target configuration C_t, an energy value can be calculated as follows:

$$E(C_t) = \left[P_{CoreStatic}(m_t, f_t) + P_{CoreDyn,s} \times \frac{V(f_t)^2}{V(f_s)^2} \right] \times T(C_t) + (MA_s + DM(w_t, w_s)) \times e_{mem} \tag{5}$$

This estimation is focused on the energy components that are affected when changing the resources considered in this study. The first component is the static core power ($P_{CoreStatic}$), which can be measured offline for different core sizes and VFs. The second component is calculated by scaling the dynamic core power during the sampled interval ($P_{CoreDyn,s}$), with the square of voltage ratio corresponding to core frequencies in C_t and C_s. We assume the availability of measurement tools that provide an online estimation of the total core power. $P_{CoreDyn,s}$ is derived by subtracting the static power ($P_{CoreStatic}(m_s, f_s)$) from this value. It should be noted that this part of the formula is simplified by neglecting the effect of core resizing on $P_{CoreDyn,s}$, which can be a source of modeling error. We explain later how we mitigate this problem. The total core power is then multiplied by the execution time derived from Equation (3) to estimate the core energy. Finally, the energy of memory accesses is added to the result. In this last component, MA_s is the total number of memory accesses during the sampled interval, $DM(w_t, w_s)$ is the difference in LLC misses between w_t and w_s, derived from the ATD, and e_{mem} is the average energy of a single memory access. Hence, in addition to the previous PMCs, we assume a counter for the total number of memory accesses.

These performance and energy models are based on simplifying assumptions that can lead to modeling errors that affect both QoS and energy savings. However, as they are designed to capture the difference between two configurations, C_t and C_s, the error reduces if these configurations are closer. For example, if the core sizes in both configurations are the same or the difference between LLC allocations is small, the accuracy of the models improves. Based on this fact, we have designed a dynamic sampling technique to mitigate the problem of modeling errors, which is presented in Section 3.6.

3.6 Phase Detection and Sampling

As explained in Section 3.5, modeling accuracy can be improved by reducing the difference between the target and the sampled configurations. This can be done by keeping a record of the measurement results over the past RM intervals as a sample set and finding the sample with the closest configuration for each modeling target. This way, modeling accuracy improves dynamically as the application progresses and more samples are collected.

However, the program behavior during an earlier interval can be substantially different from the current interval, which must be accounted for when implementing this sampling approach. The entire execution of each program can be clustered into different program *phases*, which is the topic of many previous studies. Sherwood et al. [43] present a low overhead mechanism to capture, classify, and predict phase-based program behavior during runtime. It analyzes the program basic block information captured by the program-counter value of each branch instruction along with the number of instructions executed since the last branch, which indicates the weight of that basic block. Using a hash function, different blocks (branches) are mapped into a limited number of buckets. During an execution interval, the accumulated weight of all buckets is counted, which together form a vector called a *footprint*. At each interval, the calculated footprint is compared with the past footprints stored in a table. If the Manhattan distance between two footprints is smaller than a threshold, it is classified as the same program phase. Otherwise, a new phase ID is

created for this interval. We adopt this phase detection mechanism as depicted in Figure 2(b) at the beginning of *Step-1*.

After detecting the program phase during the past interval, the sampling mechanism can store the measurement results in the "*Sampling Table*" (see Figure 2(b)). This table has an entry for every combination of program phase (Ph), core size (m), and LLC allocation (w). There is no need for sampling different core frequencies, as it can be modeled with sufficient accuracy. During *Check/Update Sampling Table*, if the past interval does not already have a corresponding entry in the table, a new entry is added. Each entry contains the core VF during sampling as well as a set of statistics required for modeling, $T_{0,s}$, $(T_{BP,s}+T_{cache,s})$, and $T_{mem,s}$ for performance modeling (see Equation (3)) plus $P_{CoreDyn,s}$ and MA_s for energy modeling (see Equation (5)).

Once the sampling table is updated based on the measurements during the past interval, the analytical models can be used with the available statistics in this table. For any target configuration C_t, the sampling table is explored for an entry with the same program phase as the past interval and the closest core size and LLC allocation. It should be noted that the RM simply predicts that the same program phase will continue in the upcoming interval. At the end of the first RM interval, the table includes only one entry that contains the measurement results for the first interval with the default baseline configuration. However, as the program execution progresses, more entries are collected in the table whenever the RM selects a new core size or LLC allocation or the program enters a new phase. This way, the modeling accuracy gradually improves across the resource configuration space. We refer to this sampling mode as *Passive* since it does not interfere with the normal operation of the RM, explained in Sections 3.3 and 3.4.

However, the proposed sampling technique supports another operating mode called *Active* sampling. In this mode, a specific configuration can be requested from the RM for a particular core whenever a new program phase is detected. The RM then forces that configuration for the upcoming interval and excludes that core from the second and third steps, that is, slack management and global optimization. This way, we can ensure that the RM finds a matching entry in the sampling table as soon as possible for that configuration. This provides a means to selectively improve the modeling accuracy for specific configurations. For example, the baseline configuration (C_{base}) is especially important since it is used as the performance target. Therefore, it must be modeled with higher accuracy to reduce the effect of modeling errors on QoS. As depicted in Figure 2(b), C_{base} is modeled early during *Step-1* because it is needed for estimating performance slack with different configurations during *Step-2*. In this study, we used only active sampling for the baseline configuration to minimize interference with the normal operation of the RM. We leave the evaluation of active sampling for other configurations for future work.

3.7 Hardware Support and Overheads

The proposed scheme imposes instruction count (IC) overheads for executing the RM algorithms and analytical modeling at each invocation. Additionally, it requires hardware support that imposes area, energy, and runtime overheads as well as memory requirements. In the rest of this section, we discuss each component in detail.

DVFS: Changing the core VF is a source of runtime overhead because of the large electrical capacitance of the supply voltage network of a processor core. Here, we assume 15 μs and 3 μJ DVFS overheads based on the pessimistic estimations reported in [34]. The simulation framework discussed in Section 4 accounts for these overheads at each DVFS decision on each core.

Core Scaling: The adaptive core architecture explained in Section 3.2 requires additional logic in each core to dynamically activate/deactivate sections of different components. Such reconfiguration can be implemented with low overhead according to previous studies on this topic. For

example, an implementation of adaptive issue queue was proposed in [6], with less than 3% gate count overhead. A similar estimation was reported in [24], with a 32-nm technology node. A core scaling operation can take up to hundreds of cycles. This is a negligible value compared with an RM interval of 100-M instructions, which is assumed in our experimental methodology.

LLC Partitioning: In order to change the partitioning of cache ways, we need to update a bit-mask per core that is $N \times W$ bits in total, where N is the number of cores and W is the number of ways in the LLC. Changing the partition sizes can lead to overheads associated with invalidation and write-backs of many cache lines. We reduce this overhead by delegating the enforcement of partition sizes to the replacement policy, thereby resulting in a gradual change over a relatively long interval (in this case, 100-M instructions). Furthermore, memory-intensive applications that benefit from such additional cache capacity allocation typically have a high MPKI value. Invalidations and write-backs impose little additional overheads on such applications.

Phase Detection: The phase detection mechanism explained in Section 3.6 requires additional hardware for each core. Based on the details provided in [43], we estimate that 32 counters with 28-bit size are needed as well as 10 registers with 24-B size for storing the footprint of each phase. This leads to a 352-B overhead plus the required logic. The runtime overhead includes a hash function and updating a counter at each branch instruction as well as about 640 operations at each RM interval. This is negligible compared with an interval size with 100-M instructions.

Performance Monitoring Counters (PMCs): As explained in Section 3.5, several PMCs are needed for the analytical models. That includes the total execution time, the stall time due to branch prediction, cache accesses, and memory accesses, as well as the total number of memory accesses. A mechanism is also needed for online measurement of total core power. Furthermore, it requires an ATD per core that operates in parallel to the main LLC. The area overhead is estimated to be less than 0.2% for a baseline 1-MiB cache in [38]. As these counters are read once at the end of each RM interval, the runtime and energy overheads are negligible.

Sampling Table: Each entry of the proposed sampling table in this work stores 5 statistics (see Section 3.6), one VF level, and a three-element key. This requires less than 10 B of memory per entry. Assuming up to 10 phases, 2 core sizes, and 32 possible LLC allocations, this table requires less than 6.25 KiB of memory per core.

Algorithm: The proposed scheme requires the execution of additional instructions to run the mentioned algorithms and analytical models. This IC overhead is added for each invocation at the end of the RM intervals. To estimate this overhead, we implemented the algorithm in C and compiled different versions for a range of core counts and when considering both fixed and adaptive core architectures. We used the Linux *perf* tool for this measurement. In addition to CSM, we have implemented the algorithm proposed in the previous work [31], which we refer to as C3P. We also extended C3P with a simple local slack management (LSM) algorithm for each core. These schemes will be discussed in detail in Section 4.5. Here, we provide the overhead estimations as a percentage of 100-M instruction intervals in Figure 3. This figure shows that the overheads remain below 1% and 0.5% up to 32 cores, assuming an adaptive and a fixed core architecture, respectively. It also shows that the overhead increment compared with the previous scheme is marginal. These overhead values are accounted for in the simulation results for every invocation of RM.

3.8 Scalability

Most of the software and hardware components described in this section — such as Algorithm 1, resource control knobs, and the analytical modeling framework — are associated with each core

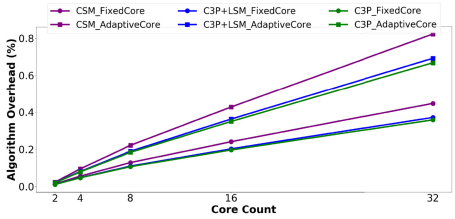


Fig. 3. IC overheads of RM schemes for different core counts.

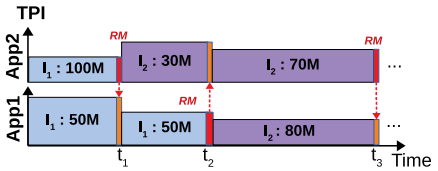


Fig. 4. Overview of the RM simulator [30, 31].

independently. Therefore, the per-core overhead does not increase for these components with a larger core count. The only component that extends to all cores is the global optimization described in Section 3.4. This component also scales linearly with the number of cores. This is evident in the overhead estimations reported in Figure 3 for up to 32 cores.

There are implementation details that must be adapted to systems with a large number of cores. For example, partitioning of cache ways will not provide sufficient granularity for such systems. In that case, other techniques, such as cache coloring [27] or Vantage [39], would be more suitable. Consequently, the analytical modeling framework must be adapted accordingly.

4 EXPERIMENTAL METHODOLOGY

In this section, we first describe the simulation framework used in this study. Next, the details of the base architecture model are presented, followed by the benchmarks and workloads. Finally, the evaluation metrics are introduced as well as the RM schemes we quantitatively compare against.

4.1 Simulation Framework

We need a simulation framework that captures the effect of the proposed RM scheme on the run-time behavior of multiple benchmarks in a multicore system. Particularly, it must be able to simulate the program phase changes on different cores when running the benchmarks to completion. Therefore, we use an in-house simulator that comprises three steps.

First, SimPoint [42] analysis is performed on benchmarks. It breaks down the full execution into consecutive intervals with a fixed instruction count, which corresponds to the RM intervals. In this study, we have selected intervals with 100-M instructions that strikes a balanced trade-off between resource management overhead and opportunities to track dynamic program behavior and phase changes. SimPoint clusters all intervals into a number of phases with similar characteristics, in this case, up to 10. Finally, it selects a representative interval from each phase for detailed simulation. It also generates a trace, called a *phase trace*, that lists the phase numbers corresponding to each interval in program order.

Next, the representative intervals of the phases for each application are simulated for the entire range of resource configurations. Each simulation consists of a 100-M instruction warm-up period followed by a 100-M instruction period of detailed simulation, using Sniper 7.2 [8] with McPAT [26]. The results of these simulations are collected in a database that is needed for the next step.

Finally, an in-house RM simulator uses the phase trace from the first step together with the simulation results database from the second step to create a proxy of a full benchmark execution when using a particular RM. The RM is invoked on each core whenever it completes one interval. To elaborate on this, we use a simplified example in Figure 4. There are two applications in this example that start from interval I_1, with the baseline resource configuration. The RM simulator

Table 1. Base Architecture Model

DVFS	Core	Global
Frequency	1–3.25 GHz (10 Steps)	2 GHz
Voltage	0.8–1.25 V (10 Steps)	1 V
Core	out-of-order, branch predictor: Pentium M type	
	L	M
Issue Width	8	4
Reorder Buffer	256	128
Reservation Station	128	64
Load Store Queue	64	32
Cache	64 B blocks, LRU replacement	
	Private	Shared
Level	L1-I/L1-D/L2	L3
Size	32/32/256 KiB	2 MiB × cores
Associativity	4/4/8	8 × cores
DVFS domain	core	global
Partition Range (4-core)	NA	2 way–32 way (256 KiB–8 MiB)
DRAM	100 ns base latency, contention queue model, 5 GB/s bandwidth per core	

Table 2. Application Categories

Cache Sensitive (CS)	Cache Insensitive (CI)
astar, bzip2, gcc, mcf, omnetpp, soplex, sphinx3, xalancbmk	GemsFDTD, cactusADM, dealII, gamess, gobmk, gromacs, h264ref, hmmer, lbm, leslie3d, libquantum, milc, namd, perlbench, povray, tonto, wrf, zeusmp

uses the database to calculate the first event (t_1), that is, the earliest time to the end of one interval in any application. After updating the runtime statistics, it invokes the RM to decide the new system configuration. The same process is repeated for the following events (t_2, t_3, ...). The RM overheads are also added to the simulation statistics at each invocation.

4.2 Base Architecture Model

Table 1 summarizes the base architecture model used in this study. We have considered two core sizes: medium (M) and large (L). The L3 cache configuration in this table is reported per each core. Furthermore, we assume a fixed DRAM bandwidth that is equally partitioned among the cores.

4.3 Workloads

We use SPEC CPU2006 benchmarks in this study. These benchmarks are divided into two categories based on cache sensitivity as follows. An application is counted as cache sensitive (CS) if both of the following conditions are true: (1) The average MPKI on the baseline LLC size (2 MiB) is greater than one. (2) When changing the LLC size around the baseline (In this case: 1-MiB → 3-MiB), the average MPKI changes by more than 20% relative to the baseline MPKI. Table 2 lists the benchmarks that belong to each category. This table does not include a few benchmarks (*bwaves*, *calculix*, and *sjeng*) because of unresolved errors during Sniper+McPAT simulations for some of the program phases.

We consider two workload scenarios. As we are mostly interested in the CS applications, in Scenario-1, different N-core workloads are generated by randomly selecting from the CS applications. This leads to high contention in LLC. The Python function `random.choice` is used for this purpose. However, for a more comprehensive evaluation, in Scenario-2, all applications in Table 2 from both categories are used to create the random workloads. We have studied 2-, 4-, 8-, and 16-core workloads to evaluate the scalability of the proposed scheme. However, due to space limitations, we do not present the results for 2-core workloads. The workloads are listed in Tables 3–5. Due to the longer simulation time, the number of workloads is reduced for systems with a larger core count.

Table 3. 4-Core Workloads

Scenario 1									
Wrk1	Wrk2	Wrk3	Wrk4	Wrk5	Wrk6	Wrk7	Wrk8	Wrk9	Wrk10
mcf, sphinx3, astar, omnetpp	mcf, mcf, sphinx3, astar	soplex, mcf, mcf, mcf	sphinx3, soplex, omnetpp, bzip2	gcc, sphinx3, xalancbmk, sphinx3	omnetpp, astar, sphinx3, sphinx3	soplex, xalancbmk, sphinx3, mcf	bzip2, gcc, gcc, mcf	gcc, astar, sphinx3, sphinx3	omnetpp, astar, xalancbmk, soplex
Scenario 2									
Wrk11	Wrk12	Wrk13	Wrk14	Wrk15	Wrk16	Wrk17	Wrk18	Wrk19	Wrk20
dealII, astar, xalancbmk, gobmk	gromacs, tonto, gromacs, xalancbmk	zeusmp, xalancbmk, milc, cactusADM	perlbench, wrf, tonto, perlbench	bzip2, GemsFDTD, libquantum, milc	sphinx3, hmmer, xalancbmk, gcc	h264ref, bzip2, soplex, mcf	sphinx3, gamess, perlbench, omnetpp	gobmk, hmmer, wrf, povray	leslie3d, namd, lbm, mcf

Table 4. 8-Core Workloads

Scenario 1				
Wrk1	Wrk2	Wrk3	Wrk4	Wrk5
xalancbmk, astar, soplex, xalancbmk, mcf, xalancbmk, soplex, astar	sphinx3, bzip2, gcc, astar, gcc, astar, gcc, astar	omnetpp, sphinx3, xalancbmk, astar, omnetpp, soplex, soplex, bzip2	omnetpp, astar, mcf, bzip2, soplex, omnetpp, omnetpp, sphinx3	mcf, mcf, gcc, gcc, bzip2, omnetpp, gcc, gcc
Scenario 2				
Wrk6	Wrk7	Wrk8	Wrk9	Wrk10
omnetpp, mcf, xalancbmk, lbm, bzip2, soplex, leslie3d, libquantum	cactusADM, namd, cactusADM, sphinx3, cactusADM, milc, zeusmp, lbm	GemsFDTD, soplex, wrf, leslie3d, gcc, astar, bzip2, wrf	zeusmp, omnetpp, dealII, gromacs, hmmer, h264ref, gobmk, GemsFDTD	gamess, libquantum, povray, gromacs, perlbench, sphinx3, tonto, sphinx3

Table 5. 16-Core Workloads

Scenario 1			
Wrk1	Wrk2	Wrk3	Wrk4
xalancbmk, soplex, bzip2, sphinx3, astar, astar, astar, gcc, xalancbmk, sphinx3, xalancbmk, mcf, mcf, astar, gcc, mcf	bzip2, sphinx3, xalancbmk, omnetpp, bzip2, gcc, soplex, bzip2, bzip2, soplex, xalancbmk, bzip2, omnetpp, bzip2, mcf, mcf	omnetpp, mcf, omnetpp, xalancbmk, mcf, mcf, mcf, soplex, bzip2, bzip2, mcf, astar, soplex, astar, astar, omnetpp	mcf, bzip2, sphinx3, astar, gcc, soplex, astar, omnetpp, mcf, gcc, soplex, astar, astar, gcc, mcf, mcf
Scenario 2			
Wrk5	Wrk6	Wrk7	Wrk8
hmmer, dealII, gromacs, astar, libquantum, dealII, zeusmp, libquantum, perlbench, xalancbmk, cactusADM, perlbench, cactusADM, bzip2, xalancbmk, omnetpp	omnetpp, tonto, tonto, milc, gromacs, bzip2, cactusADM, gobmk, lbm, xalancbmk, hmmer, namd, gobmk, bzip2, gamess, gobmk	sphinx3, milc, namd, astar, GemsFDTD, soplex, soplex, gobmk, cactusADM, h264ref, tonto, xalancbmk, libquantum, perlbench, gcc, cactusADM	cactusADM, tonto, gcc, xalancbmk, hmmer, soplex, gromacs, gcc, zeusmp, perlbench, h264ref, perlbench, wrf, hmmer, lbm, h264ref

4.4 Evaluation Metrics

In our experiments, we assume a fixed workload with no process migration or interrupt. Applications are restarted every time they finish one round of execution until the longest running application finishes its execution.

4.4.1 QoS. We assume that a QoS window is equivalent to one round of execution for each benchmark. Hence, the QoS window sizes may vary considerably from one application to another in each workload. To evaluate the QoS for a particular application, the execution time of each round is compared with the baseline execution time.

4.4.2 Energy Saving. We consider two types of energy components: (1) the energy consumption of one application, including the total core energy (with its private L1 and L2 cache) plus the dynamic energy of LLC, network-on-chip (NoC), and memory accesses, measured for the execution

of that application; and (2) the static energy of uncore components, LLC and NoC, which is application independent. In order to give equal weight to all applications, the first energy component is measured during the execution of a certain number of instructions on every core. This number is set at the beginning of the simulation according to the longest application in the workload. The second component is measured for the entire simulation. The sum of these values constitutes the total energy consumption of the system. The energy-saving is calculated by comparing this value with that of the baseline simulation, which keeps the baseline configuration the entire time.

4.5 Evaluated RM Algorithms

We have simulated and analyzed several RM approaches with different assumptions and settings. In Section 5, we compare the results for the following RM schemes:

- **C3P:** We use the RM presented in the state-of-the-art [31] for comparison. We refer to this scheme as "*Coordinated Core Configuration and Cache Partitioning*" (C3P). Given a performance constraint in terms of average time per instruction (TPI) with the baseline configuration for each core, it attempts to minimize energy by finding the best system configuration at each RM interval. Hence, C3P lacks any form of slack management. It uses the same resource control knobs as described in Section 3.2.
- **C3P+LSM**: We extend C3P with a simple form of slack management that runs independently on each core. This means that it does not affect the cache partitioning decisions made at a global level. Hence, it is called *Local Slack Management* (LSM), which operates as follows. After C3P decides a new system configuration for the upcoming interval, the LSM algorithm for each core re-evaluates the decision on core settings (VF and size) before applying the new system configuration. If sufficient performance slack is accumulated during previous intervals, it is consumed by reducing the core size/VF to save more energy.
- **CSM:** The proposed Cooperative Slack Management approach.

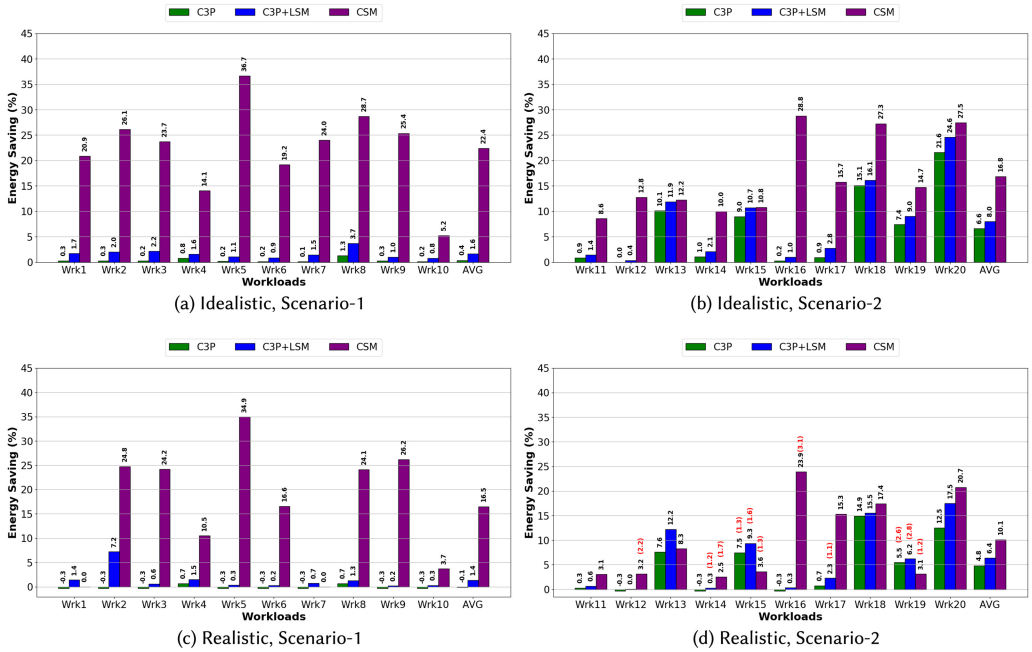
In the case of slack management (CSM & C3P+LSM), we use a lower bound on slack deposit equivalent to −0.1% of the current time at each invocation of RM compared with the beginning of the QoS window, that is, one round of benchmark execution. This creates a safe leeway for slack management to avoid considerable QoS violations at any point in time (for more details, see Section 3.3).

For a fair comparison, all RM algorithms provided earlier use the same analytical models and sampling framework presented in Sections 3.5 and 3.6. However, we also evaluate the potential of these RM schemes in an ideal scenario with no modeling errors or overheads. Therefore, the experiments are conducted with two sets of assumptions:

- **Idealistic:** Using perfect models that predict performance and energy of the upcoming interval for any configuration with no error, and assuming zero overheads.
- **Realistic:** Using the modeling framework explained in Sections 3.5 and 3.6, and counting the overheads.

5 EXPERIMENTAL RESULTS

This section evaluates the energy savings obtained by CSM in comparison with other RM schemes listed in the previous section. We start by evaluating the RM schemes with different levels of performance targets as well as a system with a fixed core architecture in Section 5.1. Next, the scalability to larger core counts is evaluated in Section 5.2. Finally, the sensitivity of CSM to different parameters, including the upper and lower bound on slack deposit and the energy budget (see Algorithm 1) is analyzed in Section 5.3.



(a) Idealistic, Scenario-1

(b) Idealistic, Scenario-2

(c) Realistic, Scenario-1

(d) Realistic, Scenario-2

Fig. 5. Energy savings with the high-performance target for different 4-core workload scenarios and modeling assumptions.

5.1 Different Baseline Configurations

To evaluate the energy savings with different levels of performance targets, we have studied two baseline core configurations in addition to a system with a fixed core architecture, as follows:

- **High-Performance:** The baseline configuration comprises a large core size and the highest frequency level (VF10) on each core, while both core sizes are supported (see Table 1).
- **Mid-range:** The baseline configuration comprises a medium core size and a mid-range frequency level (VF5), while both core sizes are supported.
- **Fixed-Core Architecture:** The system supports only the large core size while the highest VF level (VF10) is used as the baseline target.

Here, we present the experimental results for 4-core workloads with the two scenarios described in the previous section. Scenario-1 includes only cache-sensitive (CS) applications to model high contention in LLC. Scenario-2, on the other hand, considers all applications for a more comprehensive evaluation. As discussed earlier, the experiments are conducted with both *Idealistic* and *Realistic* assumptions. The former simulates perfect models with no error and ignores the overheads, while the latter uses the proposed modeling framework and accounts for the overheads. However, based on the analysis in Section 3.7, the effect of overheads is negligible, meaning that a comparison of the two establishes the potential gains from further improvements of the modeling accuracy.

5.1.1 High Performance Target. Figure 5 shows the energy-saving results with the high-performance target for different 4-core workload scenarios and modeling assumptions. The X-axis lists the workloads according to Table 3, while the Y-axis shows the energy savings. The three sets of bars from left to right represent C3P (green), C3P+LSM (blue), and CSM (purple) RM schemes.

The percentage of energy-saving is also presented on top of each bar in black. As explained in Section 4.4, the QoS of each application is evaluated based on the execution time ratio compared with the baseline. If, in any case, this ratio is greater than 101%, that is, more than 1% violation of the target, the maximum percentage of violation in the workload is shown on top of the corresponding bar in parentheses with a red color.

We now analyze the results in Figure 5. First, in workload Scenario-1, C3P does not provide a considerable improvement compared with the baseline. This is because when every application in the system is CS, the baseline LLC partitioning cannot be changed without causing any performance degradation, which is not allowed in C3P. Hence, the RM has no choice but to keep the baseline system configuration in this scenario. However, there are rare occasions during runtime when a CS application enters a short cache insensitive (CI) phase, which allows a temporary improvement in LLC partitioning and small energy savings. On the other hand, in Scenario-2, CI applications exist that can give up their baseline share to enable more energy-efficient configurations. We can see these cases in Wrk13, Wrk15, Wrk18, Wrk19, and Wrk20 in Figure 5(b), which results in an average 6.6% and up to 21.6% potential energy saving. With the realistic models, these numbers reduce to 4.8% and 14.9%, respectively, in Figure 5(d). Furthermore, modeling error leads to small QoS violations with a maximum of 2.6%, in Wrk19.

As for C3P+LSM, it provides a small improvement on top of C3P. Since C3P is limited to configurations with performance greater than or equal to the baseline target, it usually creates a small performance slack as a by-product, due to the limited range and granularity of resources. This slack can be accumulated over several intervals and consumed by LSM to save more energy. While the energy savings are still negligible for Scenario-1, C3P+LSM can potentially save up to 24.6% and on average 8% energy, in Figure 5(b). Using the realistic models, these values reduce to 17.5% and 6.4% respectively, in Figure 5(d). In this case, the modeling error can cause a maximum of 2.8% QoS violation in Wrk19.

CSM achieves significantly larger energy savings compared with both C3P and C3P+LSM. Even in Scenario-1, CSM can potentially improve the system configuration without any QoS violation. Unlike the other schemes, CSM takes advantage of small opportunities to generate more slack. As soon as sufficient slack is available for one application, it can afford to give up some of its LLC share to other CS applications. This creates more opportunities to generate slack, at a relatively low energy cost, on the applications with an increased LLC share. This creates a positive feedback loop that turns small opportunities into significant improvements over the long run. CSM can potentially save up to 36.7% (Wrk5) and, on average, 22.4% (Scenario-1) and 16.8% (Scenario-2) energy. The modeling error can reduce these values to 34.9% (Wrk5), 16.5% (Scenario-1), and 10.1% (Scenario-2), respectively. Furthermore, it can lead to a maximum QoS violation of 3.1% in Wrk16. In a couple of cases (Wrk1 and Wrk7), the modeling error entirely cancels the energy savings with CSM. When these errors cause a reduction of SD below the acceptable lower bound (Equation (2)), if CSM cannot achieve sufficient performance boost to recover, it preserves the baseline configuration to avoid considerable QoS violations.

There are a few anomalies in which modeling error leads to an improvement in energy saving. This includes Wrk2 and Wrk13 with C3P+LSM, as well as Wrk3 and Wrk9 with CSM. Due to modeling error, the RM may select a configuration that violates the performance constraint but saves more energy. However, the resulting QoS violations in these cases are negligible (less than 1%). This shows the potential for a trade-off between additional energy savings and a controlled relaxation of the performance constraints.

Modeling error does not impact all RM schemes similarly. We can see that the resulting degradation in energy savings can be larger for CSM compared with the other two schemes. This is because CSM speculatively attempts to generate slack at some energy cost to enable potential energy

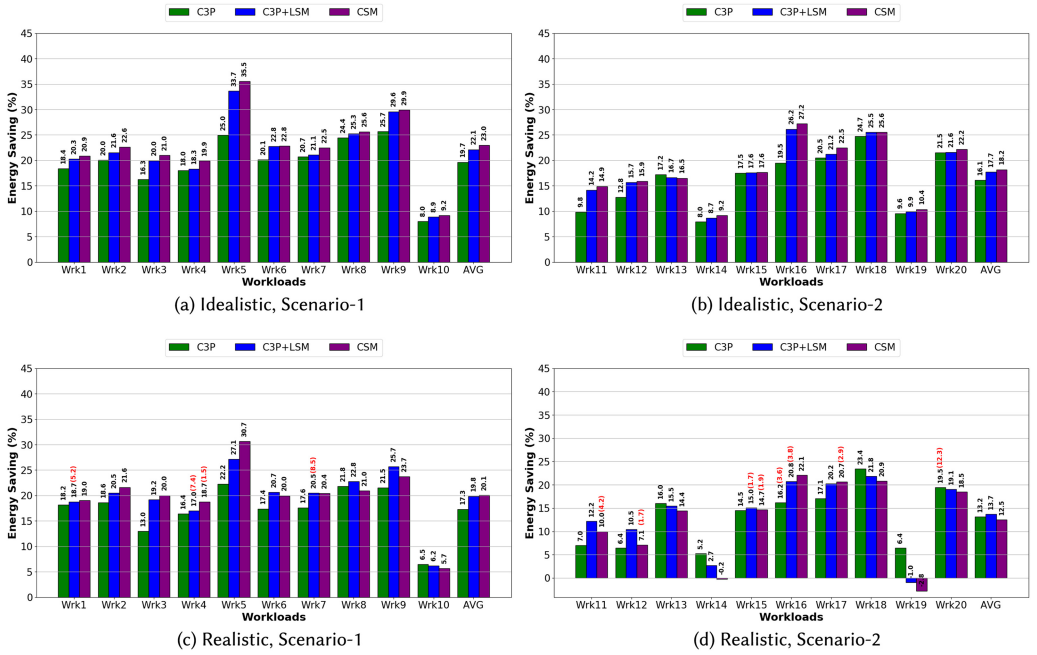


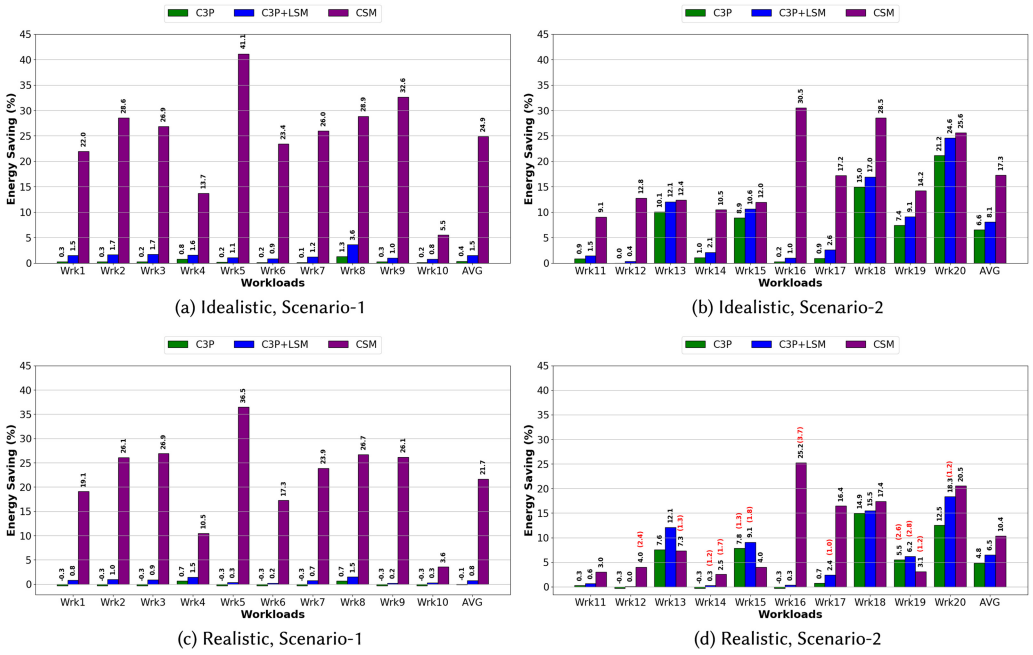
Fig. 6. Energy savings with the mid-range performance target for different 4-core workload scenarios and modeling assumptions.

savings in the future. Hence, a wrong decision due to modeling error can lead to a large energy cost. However, this can be controlled by adjusting the energy budget as explained in Section 3.3.

5.1.2 Mid-range Performance Target. Figure 6 shows the energy-saving results with a mid-range performance target. With this baseline, the large core size and higher VF levels are available to increase performance beyond the baseline target. This is similar to the baseline assumptions in [31].

We now analyze the results in Figure 6. First, energy savings with the three RM algorithms are comparable, on average. Unlike the high-performance target, in this case, C3P can potentially save up to 25% and, on average, 19.7% in Scenario-1. The reason for such an improvement is the availability of a larger core size and higher core frequency levels compared with the mid-range baseline. Instead of relying on slack, these configurations can be used to provide the required performance boost to relax the constraints on LLC partitioning. Slack management, with both C3P+LSM and CSM, can improve energy savings with a maximum of 33.7% and 35.5% and, on average, 21.1% and 23%, respectively. We see a similar trend in Scenario-2. The average energy savings for C3P, C3P+LSM, and CSM are 16.1%, 17.7%, and 18.2%, respectively.

The effect of modeling error reduces the average energy savings by a few percentage points for each RM. With the realistic models, C3P, C3P+LSM, and CSM save, on average, 17.3%, 19.8%, and 20.1% in Scenario-1 and 13.2%, 13.7%, and 12.5% in Scenario-2, respectively. However, the energy savings with CSM can be more sensitive to modeling error because of the energy costs of speculative slack generation. In Wrk14 and Wrk19, this effect causes a 0.2% and 2.8% increase in system energy with CSM. This can be controlled by adjusting the energy budget in the CSM algorithm. However, in the case of Wrk19, even C3P+LSM that does not speculatively generate slack increases system energy by 1%. In these cases, for one benchmark (*wrf*), the modeling error has led to a false



(a) Idealistic, Scenario-1

(b) Idealistic, Scenario-2

(c) Realistic, Scenario-1

(d) Realistic, Scenario-2

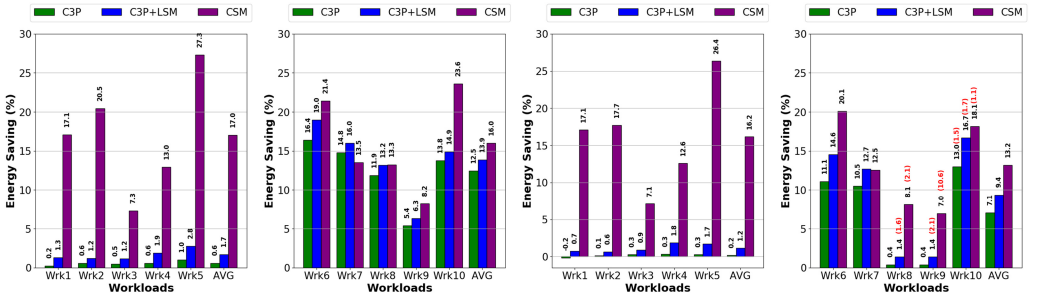
Fig. 7. Energy savings with the high-performance target and a fixed-core architecture for different 4-core workload scenarios and modeling assumptions.

negative for meeting the lower bound on SD (Equation (2)). Therefore, a faster configuration with higher energy is selected.

There are some QoS violations due to limited modeling accuracy. Compared with the high-performance target, the maximum violation has increased to 12.3% in Wrk20 with C3P, for *lbm*. The next largest violations are 8.5% in Wrk7 and 7.4% in Wrk4 using C3P+LSM. Both occurred for *sphinx3*. The main reason for such violations is an overestimation of the performance improvement by increasing the core size. To alleviate this issue, the modeling accuracy can be further improved by increasing the configurations that are sampled with the *active* mode of the proposed hybrid sampling technique (for more details, see Section 3.6).

There is an anomaly in Wrk13 in Figure 6(b) that shows a small degradation in energy savings with slack management (C3P+LSM and CSM) compared with C3P while using perfect models. This is because changing the resource configurations affects the progress of applications with respect to each other. This can change the order of program phases that appear on different cores at the same time. If this shift causes additional overlap of two cache-sensitive phases, it creates more contention in LLC and reduces the improvements with LLC partitioning.

5.1.3 Fixed-Core Architecture. As the proposed slack management framework is not fundamentally dependent on an adaptive core architecture, we decided to perform a similar evaluation with a fixed-core size. Hence, the resource control knobs, in this case, reduce to per-core DVFS and LLC partitioning. We assume the same baseline setting as the high-performance target. Figure 7 shows the results, which are very similar to the ones in Section 5.1.1. This is due to the fact that even with an adaptive architecture, the smaller core size is not likely to be selected under a high-performance target. However, when comparing the CSM results in Figures 5 and 7, we can see some improvement in energy savings using a fixed-core architecture. For example, the maximum



(a) Idealistic, Scenario-1 (b) Idealistic, Scenario-2 (c) Realistic, Scenario-1 (d) Realistic, Scenario-2

Fig. 8. Energy savings with the high-performance target for different 8-core workload scenarios and modeling assumptions.

potential energy savings increases from 36.7% to 41.1% in Wrk5, or from 25.4% to 32.6% in Wrk9. This is because there are different ways to use performance slack to save energy. Downsizing the core can linearly reduce its energy while reducing VF has a quadratic effect. Improving the LLC partitioning can also dramatically reduce system energy, especially when applications are memory intensive (Scenario-1). With a fixed core size, CSM can utilize slack only with DVFS and LLC partitioning, which can potentially save a larger amount of energy compared with core resizing.

5.2 Scalability

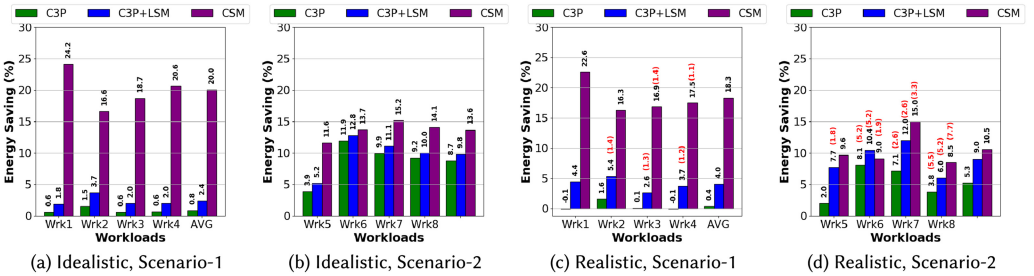
In order to study the scalability of the proposed approach, we conducted experiments similar to the high-performance target (Section 5.1.1) for the 8-core and 16-core workloads presented in Tables 4 and 5.

5.2.1 8-Core System. The 8-core simulation results are demonstrated in Figure 8, which show an overall trend similar to the 4-core experiments. CSM outperforms C3P and C3P+LSM in most cases and potentially saves up to 27.3% and, on average, 17% in Scenario-1 and 16% in Scenario-2. With realistic models, these values reduce to 26.4%, 16.2%, and 13.2%, respectively. Modeling errors also lead to a maximum QoS violation of 10.6% in Wrk9, for *hmmer*. The next largest violation is 2.1% in Wrk8. While C3P and C3P+LSM are not effective in Scenario-1, they can potentially save, on average, 12.5% and 13.9% energy in Scenario-2, respectively. Modeling errors reduce these values to 7.1% and 9.4%, respectively, while causing a maximum QoS violation of 1.5% for C3P and 2.1% for C3P+LSM.

Compared with 4-core simulations, the average results with the three RM schemes are closer in workload Scenario-2 due to the presence of more CI applications in the 8-core workloads. Therefore, C3P can find more unused cache space to optimize the LLC partitioning and save energy.

There are two anomalies in the results. First, the energy-saving with CSM is slightly smaller compared with C3P and C3P+LSM in Wrk7 in the absence of modeling errors (Figure 8(b)). The same explanation for the anomaly in Section 5.1.2 applies here. In short, there is a chance that resource management decisions may cause more cache contention by shifting the timing of program phases in different cores. Second, C3P+LSM saves more energy with realistic models compared with idealistic models in Wrk10. In this case, under the influence of modeling errors, the RM has selected configurations with lower energy that violated the performance target by a maximum of 1.7%.

5.2.2 16-Core System. Figure 9 shows the 16-core simulation results. The overall trend is consistent with 8-core and 4-core results, which demonstrates the scalability of the proposed approach. CSM can potentially save up to 24.2% and, on average, 20.0% and 13.6% energy in Scenario-1 and



(a) Idealistic, Scenario-1 (b) Idealistic, Scenario-2 (c) Realistic, Scenario-1 (d) Realistic, Scenario-2

Fig. 9. Energy savings with the high-performance target for different 16-core workload scenarios and modeling assumptions.

Scenario-2, respectively. Modeling errors reduce these values to 22.6%, 18.3%, and 10.5%, respectively. A maximum QoS violation of 7.7% is caused by modeling errors in Wrk8. While C3P is not effective in Scenario-1, it can potentially save up to 11.9% and, on average, 8.7% in Scenario-2 with idealistic models. As explained earlier, this is due to the presence of CI applications that can give up their cache allocation to CS applications. C3P+LSM slightly improves the energy savings compared with C3P. For example, the average of idealistic results increases to 9.8% in Scenario-2. In Wrk5 and Wrk7, modeling errors caused a small improvement in energy savings with C3P+LSM compared with idealistic models. This is achieved at the cost of 1.8% and 2.6% QoS violations in these workloads, respectively.

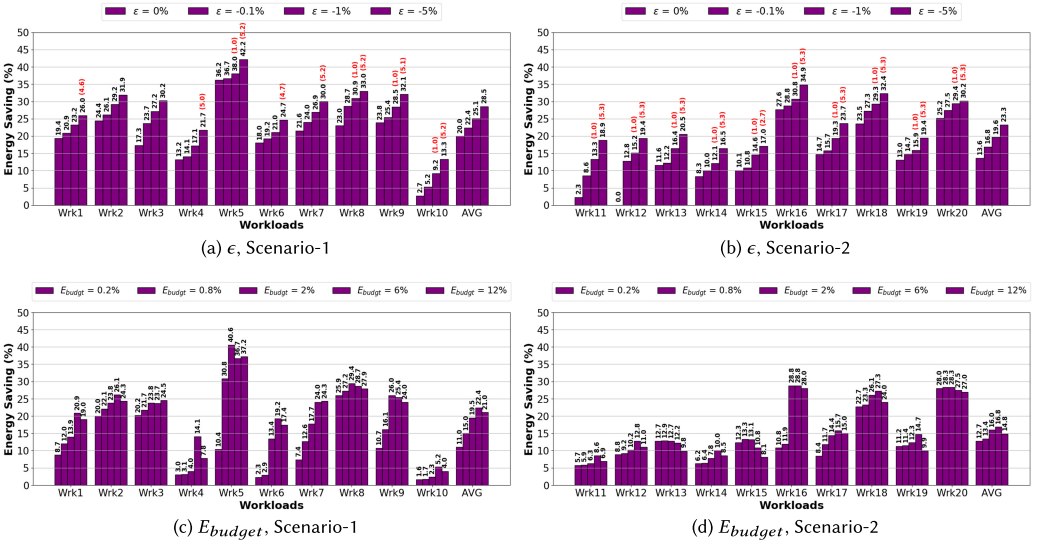
5.3 Sensitivity Analysis

There are a few predetermined parameters in CSM, including the lower and upper bounds on SD, denoted by ϵ and SD_{thr}, respectively, as well as the energy budget for generating additional slack (E_{budget}). As explained in Section 3.3, ϵ determines a minimum constraint on SD to avoid considerable QoS violations while SD_{thr} prevents additional slack generation when the current SD is already too large. These parameters are used to adjust the behavior of CSM based on conditions such as the criticality of QoS targets. Here, we provide a quantitative analysis of the sensitivity of energy-saving results to these parameters.

For this analysis, we focus on 4-core workloads with the high-performance target similar to Section 5.1.1 and *Idealistic* assumptions. By isolating the effect of modeling errors, we can see a more clear picture of the CSM behavior. According to our experiments, CSM shows negligible sensitivity to SD_{thr}, ranging from 5 ms to 2,000 ms. Therefore, we present experimental results for ϵ and E_{budget} in Figure 10. For each workload, there are multiple bars that correspond to different parameter values, as mentioned on top of the figure.

The bars in Figures 10(a) and 10(b) represent ϵ values ranging from 0% to −5% from left to right. The data show a clear trade-off between energy savings and a relaxation in the QoS constraint. By changing ϵ from 0% to −5%, the average energy savings improves from 20% to 28.5% and from 13.6% to 23.3% for workload Scenario-1 and Scenario-2, respectively. However, this can potentially cause a QoS violation close to the ϵ value. As mentioned earlier, we use $\epsilon = -0.1\%$ for the rest of the experiments.

Unlike ϵ, sensitivity to E_{budget} does not follow a fixed trend, as illustrated in Figures 10(c) and 10(d). In this case, we can see a sweet spot in the average results around $E_{budget} = 6\%$. Increasing the energy budget can raise the SD, enabling more energy-saving opportunities in future intervals. However, it imposes more energy costs for generating slack in the current interval. The combination of these two effects leads to the behavior described earlier. The sweet spot is



(a) ϵ, Scenario-1 (b) ϵ, Scenario-2

(c) E_{budget}, Scenario-1 (d) E_{budget}, Scenario-2

Fig. 10. Sensitivity of energy savings to lower bound on SD (ϵ) and energy budget (E_{budget}) for generating additional slack for different 4-core workload scenarios.

dependent on multiple factors, such as the workload characteristics and the performance targets. In the rest of the experiments, we used $E_{budget} = 6\%$ except for the mid-range performance target (Section 5.1.2), with $E_{budget} = 0.8\%$. In that case, the sweet spot shifts to smaller values because of the lower performance target. Based on this observation, we envision the potential for an upgraded version of CSM that automatically adapts the E_{budget} value at runtime.

6 RELATED WORK

This section provides a brief overview of the related work. We start by discussing the related work with respect to target resources. Next, we review different resource management approaches for supporting QoS-constrained applications, followed by studies that considered some form of slack management.

DVFS has been studied and used in numerous papers. For example, [11, 15, 20, 33, 44] used DVFS, while considering applications with QoS constraints. Core adaptation is another resource control knob that has been used, for example, in [18, 37, 40, 47] for QoS applications and in a more general scope in [1, 2, 7, 12, 23, 24, 36]. In contrast to these resources that can be controlled independently for a particular core, cache partitioning affects every core in the processor. This technique has been used in many studies, including [9, 10, 13, 14, 21, 28–32, 35, 41, 45].

We focus on the resource management approaches that include cache partitioning for QoS-constrained applications. Such applications typically require a guaranteed share of LLC. However, the initial allocation may not be utilized efficiently during runtime. To alleviate this problem, one line of research, including [10, 21, 25, 28, 35, 48], targeted for data centers, attempts to reclaim excess resources for best-effort (BE) jobs when they are not fully utilized. Such an approach can be effective when a subset of applications do not have any performance constraints and when the goal is to improve utilization or throughput. However, they are not designed for energy efficiency. Increasing the performance beyond the QoS requirements can lead to excessive energy consumption. In contrast, [33] proposed a QoS-aware task manager to improve energy efficiency in data centers. However, they did not study cache partitioning among QoS-constrained

applications. In another line of research [30–32], coordinated management of cache partitioning with core resources (DVFS and microarchitecture scaling) is proposed to reduce energy consumption while meeting the performance targets of all applications.

These studies did not consider the possibility of using slack to improve energy efficiency. Slack is typically defined in a long-term scope when a processing task finishes earlier than its deadline, for example, in [3, 4]. In that case, it can be used to slow down subsequent tasks to save energy. Thus, this form of slack is an application-dependent parameter. Furthermore, the problem of cache partitioning among QoS-constrained applications is not considered in these works. In an earlier study [9], a time-sharing approach was presented to improve cache partitioning while maintaining QoS. In this approach, applications take turns in expanding and shrinking their baseline cache partitions in order to achieve an overall improvement in throughput. This happens only if the performance boost with an expanded cache partition is considerably higher compared with the performance degradation with a shrunken partition. In contrast, management of performance slack, as presented in this article, can improve cache partitioning to save processor energy while continuously meeting the performance targets of all applications. This is achieved with the help of core resources in generating/consuming short-term performance slack that is not considered in [9]. To the best of our knowledge, such an approach has never been studied in prior work.

7 CONCLUDING REMARKS

When multiple QoS-constrained applications share a processor resource such as LLC, it may be impossible to change the baseline LLC partitioning to improve energy efficiency without causing any performance degradation. Previous work proposed coordinated management of core resources such as VF or size of the microarchitectural components together with LLC partitioning to alleviate this problem. However, continuously tracking a fixed performance target can considerably limit the energy savings. Therefore, this article presents an alternative approach based on managing short-term performance slack.

It first demonstrates that slack can be generated at a relatively low energy cost and consumed later to save a larger amount of energy. Furthermore, it shows the possibility to transfer slack from one application to another, which enables more opportunities to improve energy savings. Based on these insights, an online resource management scheme, called Cooperative Slack Management (CSM), is presented to reduce processor energy while respecting the QoS of all applications. CSM is quantitatively evaluated against the previous approach and an extension with local slack management in several different scenarios. According to the evaluations, CSM can achieve substantial energy savings, even in scenarios when the other two schemes cannot save considerable energy. CSM is especially effective when the performance target is high, the core architecture is fixed, and when there is high contention in LLC. For example, it can potentially save up to 41% energy in a case in which the savings with the other approaches is around 1%. With perfect models, the average potential energy savings range from 13% to 25% in different scenarios. However, when using the proposed modeling framework, this range reduces to 10% to 21% due to limited accuracy. That said, the proposed hybrid (active/passive) sampling technique provides a means to further improve modeling accuracy by selectively increasing the configurations that are actively sampled. This can be studied in future work.

ACKNOWLEDGMENTS

The simulations were performed on resources at Chalmers Centre for Computational Science and Engineering (C3SE) provided by the Swedish National Infrastructure for Computing (SNIC).

REFERENCES

[1] T. Adegbija and A. Gordon-Ross. 2016. Phase-based dynamic instruction window optimization for embedded systems. In *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)* (Pittsburgh, PA, USA). 397–402.

[2] D. H. Albonesi, R. Balasubramonian, S. G. Dropsbo, S. Dwarkadas, E. G. Friedman, M. C. Huang, V. Kursun, G. Magklis, M. L. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. W. Cook, and S. E. Schuster. 2003. Dynamically tuning processor resources with adaptive processing. Computer (2003).

[3] M. W. Azhar, M. Pericàs, and P. Stenström. 2019. SaC: Exploiting execution-time slack to save energy in heterogeneous multicore systems. In *Proceedings of the 48th International Conference on Parallel Processing (Kyoto, Japan) (ICPP'19)*. Association for Computing Machinery, 1–12.

[4] M. W. Azhar, P. Stenström, and V. Papaefstathiou. 2017. Sloop: QoS-supervised loop execution to reduce energy on heterogeneous architectures. *ACM Transactions on Architecture and Code Optimization* 14, 4 (2017), 1–25.

[5] B. Bowhill, B. Stackhouse, N. Nassif, Z. Yang, A. Raghavan, C. Morganti, C. Houghton, D. Krueger, O. Franza, J. Desai, J. Crop, D. Bradley, C. Bostak, S. Bhimji, and M. Becker. 2015. 4.5 The Xeon processor E5-2600 v3: A 22nm 18-core product family. *IEEE Journal of Solid-State Circuits* 51, 1 (2015), 92–104.

[6] A. Buyuktosunoglu, D. Albonesi, S. Schuster, D. Brooks, P. Bose, and P. Cook. 2001. A circuit level implementation of an adaptive issue queue for power-aware microprocessors. In *Proceedings of the 11th Great Lakes symposium on VLSI (West Lafayette, Indiana, USA)*. Association for Computing Machinery, 73–78.

[7] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. W. Cook, and D. H. Albonesi. 2001. An adaptive issue queue for reduced power at high performance. In *International Workshop on Power-Aware Computer Systems (Berlin, Heidelberg)*. Springer, 25–39.

[8] T. E. Carlson, W. Heirman, S. Eyerman, I. Hur, and L. Eeckhout. 2014. An evaluation of high-level mechanistic core models. *ACM Transactions on Architecture and Code Optimization* 11, 3 (2014), 1–25.

[9] J. Chang and G. S. Sohi. 2007. Cooperative cache partitioning for chip multiprocessors. In *ACM International Conference on Supercomputing 25th Anniversary Volume (Munich, Germany)*. Association for Computing Machinery, 402–412.

[10] S. Chen, C. Delimitrou, and J. F. Martínez. 2019. PARTIES: QoS-aware resource partitioning for multiple interactive services. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS) (Providence, RI, USA)*. Association for Computing Machinery, 107–120.

[11] K. Choi, K. Dantu, W.-C. Cheng, and M. Pedram. 2002. Frame-based dynamic voltage and frequency scaling for a MPEG decoder. In *Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design (New York, NY, USA)*. Association for Computing Machinery, 732–737.

[12] Y. Eckert, S. Manne, M. J. Schulte, and D. A. Wood. 2012. Something old and something new: P-states can borrow microarchitecture techniques too. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design (Redondo Beach, California, USA)*. Association for Computing Machinery, 385–390.

[13] X. Fu, K. Kabir, and X. Wang. 2011. Cache-aware utilization control for energy efficiency in multi-core real-time systems. In *2011 23rd Euromicro Conference on Real-Time Systems (Porto, Portugal)*. 102–111.

[14] L. Funaro, O. A. Ben-Yehuda, and A. Schuster. 2016. Ginseng: Market-driven LLC allocation. In *2016 USENIX Annual Technical Conference (USENIX ATC 16) (Denver, CO)*. USENIX Association, 295–308.

[15] M. Ghorbani Moghaddam and C. Ababei. 2017. Dynamic energy management for chip multi-processors under performance constraints. *Microprocessors and Microsystems* 54 (2017), 1–13.

[16] M. S. Gupta, G.-Y. Wei, and D. Brooks. 2008. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture (Salt Lake City, UT, USA)*. 123–134.

[17] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer. 2016. Cache QoS: From concept to reality in the Intel ® Xeon ® processor E5- 2600 v3 product family. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA) (Barcelona, Spain)*. 657–668.

[18] C. J. Hughes, J. Srinivasan, and S. V. Adve. 2001. Saving energy with architectural and frequency adaptations for multimedia applications. In *Proceedings 34th ACM/IEEE International Symposium on Microarchitecture MICRO-34 (Austin, TX, USA)*. IEEE, 250–261.

[19] R. Jevtic, H.-P. Le, M. Blagojevic, S. Bailey, K. Asanovic, E. Alon, and B. Nikolic. 2015. Per-core DVFS with switched-capacitor converters for energy efficiency in manycore processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 23, 4 (2015), 723–730.

[20] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez. 2015. Rubik: Fast analytical power management for latency-critical systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO) (Waikiki, HI, USA)*. 598–610.

[21] H. Kasture and D. Sanchez. 2014. Ubik: Efficient cache sharing with strict QoS for latency-critical workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (Salt Lake City, Utah, USA)*, Vol. 49. Association for Computing Machinery, 729–742.

[22] S. K. Khatamifard, L. Wang, W. Yu, S. Köse, and U. R. Karpuzcu. 2017. ThermoGater: Thermally-aware on-chip voltage regulation. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA) (Toronto, ON, Canada)*. 120–132.

[23] M. Khavari Tavana, M. H. Hajkazemi, D. Pathak, I. Savidis, and H. Homayoun. 2018. ElasticCore: A dynamic heterogeneous platform with joint core and voltage/frequency scaling. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26, 2 (2018), 249–261.

[24] Y. Kora, K. Yamaguchi, and H. Ando. 2013. MLP-aware dynamic instruction window resizing for adaptively exploiting both ILP and MLP. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (Davis, California)*. Association for Computing Machinery, 37–48.

[25] N. Kulkarni, G. Gonzalez-Pumariega, A. Khurana, C. A. Shoemaker, C. Delimitrou, and D. H. Albonesi. 2020. CuttleSys: Data-driven resource management for interactive services on reconfigurable multicores. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO) (Athens, Greece)*. 650–664.

[26] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. 2009. McPAT 1.0: An integrated power, area, and timing modeling framework for multicore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (New York, NY, USA)*. Association for Computing Machinery, 469–480.

[27] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. 2008. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture (Salt Lake City, UT)*. 367–378.

[28] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. 2015. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*. Association for Computing Machinery, New York, NY, USA, 450–462.

[29] M. Moreto, F. J. Cazorla, A. Ramirez, R. Sakellariou, and M. Valero. 2009. FlexDCP: A QoS framework for CMP architectures. *ACM SIGOPS Operating Systems Review* 43, 2 (2009), 86–96.

[30] M. Nejat, M. Manivannan, M. Pericàs, and P. Stenström. 2020. Coordinated management of DVFS and cache partitioning under QoS constraints to save energy in multi-core systems. *Journal of Parallel and Distributed Computing (JPDC)* 144 (2020), 246–259.

[31] M. Nejat, M. Manivannan, M. Pericàs, and P. Stenström. 2020. Coordinated management of processor configuration and cache partitioning to optimize energy under QoS constraints. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS) (New Orleans, LA, USA)*. IEEE, 590–601.

[32] M. Nejat, M. Pericàs, and P. Stenström. 2019. QoS-Driven coordinated management of resources to save energy in multi-core systems. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS) (Rio de Janeiro, Brazil)*. IEEE, 303–313.

[33] R. Nishtala, V. Petrucci, P. Carpenter, and M. Sjalander. 2020. Twig: Multi-Agent task management for colocated latency-critical cloud services. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA) (San Diego, CA, USA)*. 167–179.

[34] S. Park, J. Park, D. Shin, Y. Wang, Q. Xie, M. Pedram, and N. Chang. 2013. Accurate modeling of the delay and energy overhead of dynamic voltage and frequency scaling in modern microprocessors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 5 (2013), 695–708.

[35] T. Patel and D. Tiwari. 2020. CLITE: Efficient and QoS-aware co-location of multiple latency-critical jobs for warehouse scale computers. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA) (San Diego, CA, USA)*. 193–206.

[36] P. Petoumenos, G. Psychou, S. Kaxiras, J. M. Cebrian Gonzalez, and J. L. Aragon. 2010. MLP-aware instruction queue resizing: The key to power-efficient performance. In *International Conference on Architecture of Computing Systems (Berlin, Heidelberg)*. Springer, 113–125.

[37] R. P. Pothukuchi, A. Ansari, P. Voulgaris, and J. Torrellas. 2016. Using multiple input, multiple output formal control to maximize resource efficiency in architectures. In *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA) (Seoul, Korea (South))*. IEEE, 658–670.

[38] M. Qureshi and Y. Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06) (Orlando, FL, USA)*. 423–432.

[39] D. Sanchez and C. Kozyrakis. 2011. Vantage: Scalable and efficient fine-grain cache partitioning. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (San Jose, California, USA) (ISCA'11)*. Association for Computing Machinery, 57–68.

[40] R. Sasanka, C. J. Hughes, and S. V. Adve. 2002. Joint local and global hardware adaptations for energy. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (San Jose, California) (ASPLOS X)*. Association for Computing Machinery, 144–155.

[41] A. Sharifi, S. Srikantaiah, A. Mishra, M. Kandemir, and C. Das. 2011. METE: Meeting end-to-end QoS in multicores through system-wide resource management. In *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems (San Jose, California, USA)*. Association for Computing Machinery, 13–24.

[42] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. 2002. Automatically characterizing large scale program behavior. *ACM SIGARCH Computer Architecture News* 37, 10 (2002), 45–57.

[43] T. Sherwood, S. Sair, and B. Calder. 2003. Phase tracking and prediction. In *Proceedings of the 30th Annual International Symposium on Computer Architecture (San Diego, California) (ISCA'03)*. Association for Computing Machinery, 336–349.

[44] J. Suh, C.-T. Huang, and M. Dubois. 2015. Dynamic MIPS rate stabilization for complex processors. *ACM Transactions on Architecture and Code Optimization (TACO)* 12, 1 (2015), 1–25.

[45] N. Takagi, H. Sasaki, M. Kondo, and H. Nakamura. 2009. Cooperative shared resource access control for low-power chip multiprocessors. In *Proceedings of the 2009 ACM/IEEE International Symposium on Low Power Electronics and Design (San Fancisco, CA, USA)*. Association for Computing Machinery, 177–182.

[46] B. Wolford, T. Speier, and D. Bhandarkar. 2017. Qualcomm Centriq 2400 processor. In *Hot Chips: A Symposium on High Performance Chips (HC29)*.

[47] Y. Zhou, H. Hoffmann, and D. Wentzlaff. 2016. CASH: Supporting IaaS customers with a sub-core configurable architecture. In *Proceedings of the 43rd International Symposium on Computer Architecture (Seoul, Republic of Korea) (ISCA'16)*. IEEE Press, 682–694.

[48] H. Zhu and M. Erez. 2016. Dirigent: Enforcing QoS for latency-critical tasks on shared multicore systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (Atlanta, Georgia, USA) (ASPLOS'16)*. Association for Computing Machinery, 33–47.

Received July 2021; revised October 2021; accepted December 2021