ARCTIC: Approximate Real-Time Computing in a Cache-Conscious Multicore Environment

Sangeet Saha§, Shounak Chakraborty§, Sukarn Agarwal, Magnus Själander, and Klaus D. McDonald-Maier

***Abstract*—Improving result-accuracy in approximate computing (AC) based time-critical systems, without violating power constraints of the underlying circuitry, is gradually becoming challenging with the rapid progress in technology scaling. The execution span of each AC real-time tasks can be split into a couple of parts: (i) the mandatory part, execution of which offers a result of acceptable quality, followed by (ii) the optional part, which can be executed partially or completely to refine the initially obtained result in order to increase the result-accuracy, while respecting the time-constraint. In this article, we introduce a novel hybrid offline-online *scheduling strategy*, ARCTIC, for AC real-time tasks. The goal of real-time scheduler of ARCTIC is to maximise the results-accuracy (QoS) of the task-set with opportunistic shedding of the optional part, while respecting system-wide constraints. During execution, ARCTIC retains exclusive copy of the private cache blocks only in the local caches in a multi-core system and no copies of these blocks are maintained at the other caches, and improves performance (i.e., reduces execution-time) by accumulating more live blocks on-chip. Combining offline scheduling with the online cache optimization improves both QoS and energy efficiency. While surpassing prior arts, our proposed strategy reduces the task-rejection-rate by up to 25%, whereas enhances QoS by 10%, with an average energy-delay-product gain of up to 9.1%, on an 8-core system.**

***Index Terms*—Real-time systems, Approximate Computing, QoS Improvement, Energy Efficiency, Cache Management**

I. INTRODUCTION

In time-constrained systems, functional correctness not only depends on the result-accuracy, but also that results are produced before given deadlines [34]. For such real-time scenarios, approximated result obtained before the deadline is preferable over an accurate result produced after the deadline. In a plethora of application areas, such as multimedia processing, tracking of mobile targets, real-time heuristic search, information gathering and control systems, an approximate result, obtained within the time-limit is usually acceptable [6]. For example, in the case of video streaming applications, frames with lower quality are preferable over completely missing frames. In the case of target tracking, an approximated estimation of the target's location produced within the deadline is preferred to an accurate location, obtained too late. In such domains, applications are usually modeled as real-time task graphs, or precedence constrained task graphs (PTGs), whose nodes denote application tasks and edges denote inter-task dependencies. Further, each of these tasks is logically decomposed into a mandatory part and an optional part [11], [33], [35].

The entire mandatory part has to be completed prior to the deadline to generate minimally acceptable QoS, followed by a partial/complete execution of the optional part, subject to availability of system resources, to improve accuracy of the initially obtained result within the deadline. The QoS increases with the number of execution cycles spent on the optional part, and based upon the amount of execution cycles of the optional part, a task can offer various QoS levels. However, it is worth noting that, compared to a lower QoS level, a higher QoS level demands additional computation, which consequently leads to higher execution time and associated energy overheads. Thus, the desire for enhanced QoS with improved energy efficiency and the completing tasks within their deadlines are often in conflict with each other. Hence, *given a real-time application modeled as a PTG, where the tasks have multiple QoS levels and need to be scheduled on a multiprocessor platform, maximising system-level QoS by allocating appropriate quality levels, cores, and execution start times to the tasks, and assigning core frequency, while satisfying all timing, power, precedence, and resource-related constraints, is a challenging scheduling problem.*

Approximate real-time scheduling techniques are often classified as offline or online, based on whether scheduling decisions are made at design time or at runtime [17]. Prior art considered approximate computing based offline real-time task scheduling [11], [35] by focusing on the processor cores from a power-performance perspectives in case of chip multiprocessor (CMP) based systems. As time-constrained systems require a high degree of timing predictability, it is typically preferable to employ offline scheduling algorithms for such systems, as this allows all timing requirements to be specified offline before execution [42]. However, rigid adherence to static schedules during online execution could result in reduced performance, if the runtime architectural characteristics are not considered [12].

Architectural techniques are effective in improving the energy efficiency of the last level caches (LLCs) [13]–[15], [38]. A prior coherence management technique for shared LLC, proposed by Lodde et al., shrinks LLC size dynamically at the way level granularity, while maintaining only copies of the private blocks in the local cache [30]. Later Albericio et al. proposed a "reuse cache" that decouples tag and data for unused LLC blocks [4] and improves performance, while

S.Saha and K. McDonald-Maier are with the Embedded and Intelligent Systems Lab, University of Essex, Colchester, UK. S. Chakraborty and M. Själander is with the Department of Computer Science, Norwegian University of Science and Technology, Trondheim, Norway. S. Agarwal is with the School of Computing and Electrical Engineering, IIT Mandi, India.

e-mail: (sangeet.saha@essex.ac.uk, shounak.chakraborty@ntnu.no, sukarn@iitmandi.ac.in, magnus.sjalander@ntnu.no, kdm@essex.ac.uk).

§Equal contribution

incurring 16% additional LLC storage, associated with a noticeable power overhead. ARCTIC introduces a critical management of the private blocks in a shared LLC to improve both power efficiency and performance, by increasing live-block count in the LLC, without incurring noticeable overheads. We argue and empirically validate that our novel runtime architectural technique to improve performance as well as energy efficiency can also generate a slack time before the deadline by reducing the task's execution span. Such slacks can also be exploited either to maximise QoS by executing more from the optional part of the task or to enhance energy efficiency of the underlying hardware by enabling power gating mode [37]. However, *it is worth noting that, even in these situations, static schedules are still crucial as they can serve as the basis for further online QoS stimulation considering runtime characteristics of the task.*

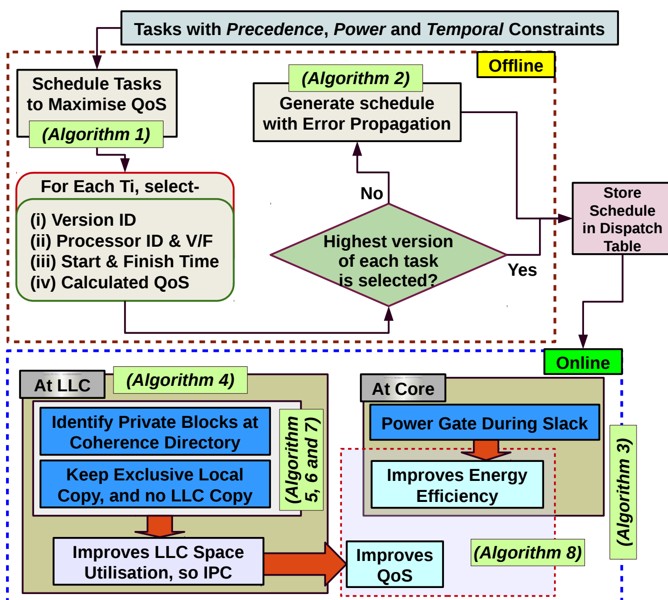


Fig. 1: ARCTIC: Process Overview

In this paper, we propose a scheduling technique, ARCTIC, for approximate real-time task-set that generates a schedule in offline with an objective to maximise the QoS, which is further enhanced along with significant energy reduction at runtime. Our considered task-set is dependent and hence can be represented by a PTG (Sec. III), are scheduled on a CMP, where each task can have multiple versions with distinct degrees of accuracy. The entire concept of ARCTIC is shown in Figure 1, where the upper half of the figure depicts the offline part and the lower part elaborates the online part. Respective algorithms in Figure 1 for individual sub-stages of ARCTIC-*Offline* and ARCTIC-*Online* which will be elaborated in Sec. IV and V, respectively.

Our key contributions can be summarized as follows:

- ARCTIC-*Offline* is a **power constrained based task-scheduling** strategy (Sec. IV) that is formulated initially as an optimization problem to maximise QoS, which is solved to generate an optimal schedule. Each task is assigned to a processor core along with a voltage and frequency setting.
- In cases where tasks are scheduled with lower accuracy, the portions of the optional parts that have not been scheduled are added with the successor task(s) to enhance QoS. This phenomenon is termed as **Error Propagation**, which might cause a deadline violation, that is however ameliorated by **readjusting the voltage and frequency** at task level granularity, without violating the power-budget (Sec. IV).
- At runtime, ARCTIC-*Online* maintains **exclusive copies of the private LLC blocks** in the respective local caches, and thus frees up space in the shared LLC (Sec. V), that are exploited to accumulate more live LLC blocks to improve performance.
- The **slacks generated by reduced execution-times are exploited** either to enhance QoS by executing more from optional parts of the tasks, or to improve energy efficiency by turning off the cores (Sec. V).

While surpassing prior arts, ARCTIC-*Offline* reduces the task-rejection-rate by up to 25%, whereas ARCTIC-*Online* enhances QoS further, by 10%, with an average energy-delay-product (EDP) gain of up to 9.1%, for an 8-core based CMP (Sec. VI).

II. Relevant Prior Art

In recent years, researchers have developed various approaches for the real-time scheduling of PTGs on multicore platforms. One set of the approaches are focused on energy-minimisation. Since many of the CMP-based real-time systems are battery-powered, reducing energy in such systems has become an active research topic in recent past [5], [36]. For energy savings, DVFS (dynamic voltage frequency scaling) and DPM (dynamic power management) are typically employed. Scheduling time-constrained dependent tasks on the CMP platform by considering the energy/power cap is however becoming challenging with technology scaling [8]. An ILP-based optimal approach was employed to solve the problem in a prior art [16]. However, incorporating DVFS with task allocation and scheduling decision is not straightforward, as additional variables regarding the core selection, data dependency, and Voltage/frequency (V/F) selection are required. Moreover, these variables are interconnected, which makes the problem challenging. To tackle this problem, researchers applied system-level DVFS, where the frequency of each processor was not adjusted individually. But, this technique is not suitable for time-constrained systems. Researchers attempted to build up sub-optimal energy-aware scheduling strategies for real-time task sets with different system-wide constraints [23], [32], [45].

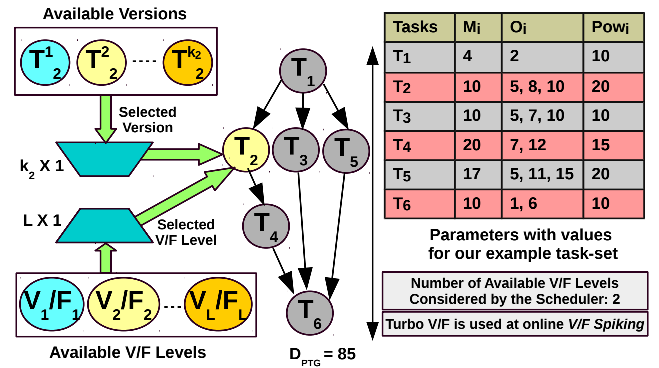
In 2013, Kim considered QoS-based real-time scheduling problem to improve the overall QoS under task deadlines [24]. Subsequently, research into QoS-aware real-time scheduling has emerged. The concept of approximate computing to meet the energy budget of a large-scale real-time system was introduced for independent tasks [11]. Zhou et al. [47] took energy consumption into account and proposed an energy-adaptive and QoS-driven task mapping method. The energy-efficient scheduling of the dependent approximate tasks was

considered in some prior arts [29], [35] with DVFS at the cores. However, these methods did not consider the effect on system performance when QoS degradation occurs for a task, and the proposed optimal approaches also suffer from high computational complexity. In some recent works [12], [38], thermal efficient task scheduling for dependent approximate real-time tasks was discussed, where the QoS offered in offline mode is further enhanced by employing runtime architectural techniques. In ARCTIC, a low-complexity optimal offline task scheduling technique is proposed to maximise QoS. This obtained QoS is further enhanced by incorporating runtime architectural parameters. Moreover, to realistically frame the behavior of QoS-aware systems, we introduce the concept of error propagation.

Improving QoS as well as energy efficiency by employing runtime architectural techniques can be a viable option to be considered in approximate real-time computing. To reduce leakage power and area occupancy of the conventional MOSFET-based LLC, Lodde et al. proposed a decoupled technique for data and tag entries of the LLC [30]. The private blocks are placed only in the local cache, and LLC data array is trimmed to be used only for the shared blocks. This technique significantly saves leakage power, but at the cost of performance loss up to 10%. Later, Albericio et al. proposed a tag/data decoupled technique that keeps both data and tag only for the blocks that have been reused [4]. This policy leads to a 16% increase in (MOSFET-based) LLC storage capacity; however, authors did not consider the power consumption of the LLC. In a recent study, researchers exploit dataless LLC entries to overcome several write issues in an NVM-based hybrid LLC [3]. Another recent approach attempted to reduce refresh counts in an embedded-DRAM-based LLC by segregating the private blocks and decoupled tag and data for such blocks [31]. A coherence protocol based private block management was also proposed [39], which incurs overhead to the coherence management. Most of these prior policies limit usages of the data array either to reduce leakage consumption of the MOSFET-based LLC or to overcome writing issues of the NVM caches. In ARCTIC, we considered FinFET-based LLC that consumes lower leakage than the MOSFET-based LLC and do not have writing issues like NVMs. Hence, in ARCTIC, by managing the private blocks, the freed-up LLC space is exploited to enhance the overall performance.

III. System Model and Assumptions

Our considered CMP consists of m homogeneous cores, denoted as $P = P_1, P_2, ..., P_m$. Each core can support L distinct V/F settings, denoted as $V = V_1, V_2, ..., V_L$ and $F = F_1, F_2, ..., F_L$, where $V_i < V_{i+1}$ and $F_i < F_{i+1}$. Our considered application, which is collection of a set of sub-applications (also termed tasks in this paper), can be represented as a PTG (see Figure 2), $G = (T, E)$, where T is a set of nodes $(T = T_i | 1 \leq i \leq n)$ and E is a set of directed edges $(E = \{\langle T_i, T_j \rangle \mid 1 \leq i, j \leq n; i \neq j\})$. Each node T_i denotes a task of the PTG, and each directed edge specifies an execution order that the T_j can only start after the task T_i is completed. Being a real-time application, G has to be executed



Tasks	Mi	Oi	Powi
T1	4	2	10
T2	10	5, 8, 10	20
T3	10	5, 7, 10	10
T4	20	7, 12	15
T5	17	5, 11, 15	20
T6	10	1, 6	10

Fig. 2: Precedence task graph (PTG) with timing parameters and power requirements.

within the given deadline, by executing all the associated task (T_i). The end-to-end application deadline is assumed to be D_{PTG}. We also assume that, T_i can have k_i different versions, $T_i = T_i^1, T_i^2, ..., T_i^{k_i}$, those are distinct by their respective execution lengths (O_i), denoted as $O_i^1, O_i^2, ..., O_i^{k_i}$, where O_i^p offers higher result-accuracy than O_i^q , if $p > q$. For each optional part of a task (O_i), there exists a separate executable module, that is executed after execution of the mandatory portion (M_i) of the respective task, T_i. The length of the j-th version of task T_i (len_i^j) can be defined as: $len_i^j = M_i + O_i^j$. Note that, len_i^j includes the cycles required for accessing LLC, which we obtain by executing an individual task for a particular configuration. We define result-accuracy Acc_i^j of T_i^j as the executed optional part of the task, O_i^j (i.e., $Acc_i = O_i^j$). Thus, the overall system level result-accuracy (QoS) is now defined as the sum of the executed cycles of O_i^j for all the tasks [11], which can be represented as: $QoS = \sum_{i=1}^{n} O_i^j \mid T_i = T_i^j$.

IV. ARCTIC-*Offline*

With an objective to maximise the QoS, subject to the system-wide constraints, ARCTIC-*Offline* allocates tasks to the processor cores with the assigned V/F. Our scheduling problem is bounded by a set of constraints and the problem is detailed in Sec. IV-A. We further discuss the concept of error propagation in Sec. IV-B, before concluding the section with a working example.

A. ARCTIC: ILP based Scheduling

We present a scheduling strategy based on integer linear programming (ILP). For this purpose, we first introduce an integer decision variable $S_i \in \mathbb{Z}^+$ to capture the start time of each task T_i, where $\mathbb{Z}^+$ denotes the set of positive integers. We further define a binary decision variable, $Z_{ikl\theta}$, where, $i = 1, 2, ..., n$; $k = 1, 2, ..., k_i$; $l = f_1, f_2, ..., f_L$; and $\theta = 1, 2, ..., m$; Here indices, i, k, l, and θ denote task ID, corresponding version ID, viable frequency of the θ-th processor and the processor ID, respectively. $Z_{ikl\theta} = 1$, if the k-th version of T_i (i.e. T_i^k) executes on processor θ at frequency f_l, otherwise 0. We define another binary variable Y_{ij}, where $Y_{ij} = 1$, if task T_i starts before T_j, else 0. If a

task T_i is executed with the kth version at frequency level l, then its execution time will be denoted as $ET(i,k,l)$. We now present the objective function with constraints on the binary variables to model the scheduling problem.

$$\textbf{Maximise} \quad \textbf{QoS} \tag{1a}$$

$$QoS(\mathcal{A}) = \sum_{\theta=1}^{m}\sum_{i=1}^{n}\sum_{k=1}^{k_i}\sum_{l=f_1}^{f_L} Z_{ikl\theta} \times O_i^k \tag{1b}$$

Subject to:

$$\sum_{k=1}^{k_i}\sum_{l=f_1}^{f_L}\sum_{\theta=1}^{m} Z_{ikl\theta} = 1 \tag{1c}$$

$$S_n + \sum_{k=1}^{k_n}\sum_{l=f_1}^{f_L}\sum_{\theta=1}^{m}(ET(n,k,l) \times Z_{nkl\theta}) - 1 \leq D_{PTG} \tag{1d}$$

$$Pow_{peak} = max\{Pow_{sys}\} \tag{1e}$$

$$Pow_{peak} \leq Pow_BGT \tag{1f}$$

$$\forall(\langle T_i, T_j\rangle) \in E,\ S_i + \sum_{k=1}^{k_i}\sum_{l=f_1}^{f_L}\sum_{\theta=1}^{m}((ET(i,k,l) \times Z_{ikl\theta}) \leq S_j \tag{1g}$$

$$Y_{ij} + Y_{ji} > 0 \tag{1h}$$

$$Y_{ij} + Y_{ji} \leq 1 \tag{1i}$$

$$S_i + \sum_{k=1}^{k_i}\sum_{l=f_1}^{f_L}\sum_{\theta=1}^{m}((ET(i,k,l) \times Z_{ikl\theta}) \leq S_j + (1 - Y_{ij}) \times \alpha \tag{1j}$$

Equation 1b presents the objective function in the above formulation, whereas Equation 1c enforces the constraint that each task T_i is assigned to exactly one processor with a particular version and executed at one frequency level. The application must meet its end-to-end absolute deadline D_{PTG}. Hence, the sink node T_n should be finished by D_{PTG}, which is represented by the constraint enforced in Equation 1d. The system-wide power constraint must be satisfied i.e. the peak power consumption of the defined system must not exceed the stipulated power budget. We represent the power constraints in Equation 1e and 1f, where Pow_{peak} represents the peak power consumption of the system. On the other hand, Pow_{sys} includes dynamic plus static power consumption of all the busy processors and is the summation of power consumption of all the tasks executing at that instant. Equation 1g, the precedence constraint between the tasks (T_i & T_j), ensures that the execution of T_j should commence only after the completion of its predecessor T_i. Equation 1h to 1j represent the necessary constraints in order to avoid time-wise overlapping between tasks executing at the same processor. If tasks are executed in the opposite order, we use α nullification to deactivate the constraint where, α represents a large value.

B. ARCTIC: Error Propagation

As per the ILP-based scheduling, when a task (T_i) is scheduled with a lower version (assuming ζ-th version, where $1 < \zeta < k_i$), the result of a partially completed task shows

Algorithm 1: Generating Schedule and V/F Readjustment

Input:
i. Task graph $G(T, E)$
ii. k_i: Number of versions of each task T_i
iii. len_i^j: Execution length of jth version of task T_i
iv. D_{PTG}: The deadline of the task graph.
v. Acc_i^j: accuracy achieved by executing j^{th} version of T_i
vi. F_l^j: implies a frequency level (l) for processor j.
Output:
i. Task Schedule /* Selected Task versions (ζ_i), Execution start times (st_i), Mapped Processor id: (P_i^j i.e. i^{th} task on j^{th} Processor, Obtained Accuracy) */
ii. Achieved system-level QoS.

1. By considering all $T_i \in T$ and all processors P, a schedule will be generated by employing the ILP given in Equation 1a to 1j, and this schedule is named as $PrimarySched$;
2. /* Our ILP based scheduling mechanism will map the tasks to the cores and also selects the version of each task that will be executed. */
3. **for** *each* $T_i \in T$ **do**
4.   Generate the list of predecessor (PAR_i, list of predecessors for T_i) for each of the $|T| - 1$ tasks, as our single source task does not have any predecessor;
5.   **if** T_i *is not mapped/selected with its highest version* **then**
6.   Calculate OE_i and subsequently IE_i by employing Equation 2 and 3, respectively;
7.   Derive EM_j for T_j through Equation 4, where T_j is a successor of T_i;
8. By considering EM_i for each T_i, modify $PrimarySched$, and this modified schedule is named as $SchedMod$;
9. Let us assume task $T_\eta^{P_j}$ is the last task scheduled at processor P_j ($\forall P_j \in P$) in our schedule and one of such $T_\eta^{P_j}$ will trivially be the sink task;
10. **if** $\exists\ T_\eta^{P_j}$*, where* $FinishTime(T_\eta^{P_j}) > D_{PTG}$ **then**
11. $SchedNew$ = Exploit_Idle_Slot($SchedMod$) (Call Algorithm 2);
12.   **if** $T_\eta^{P_j} \in SchedNew,\ FinishTime(T_\eta^{P_j}) \leq D_{PTG}$ **then**
13. $SchedFinal$ = $SchedNew$;
14.   **else**
15.   Discard the schedule and return ;
16. **else**
17. $SchedFinal$ = $SchedMod$;
18. Store $SchedFinal$ in dispatch table and return QoS;

lower accuracy than the maximum possible value and this difference is termed output error (OE_i) and is represented:

$$OE_i = O_i^{k_i} - O_i^{\zeta} \tag{2}$$

Since, the tasks (T_i) are dependent, the error OE_i of T_i is propagated to its successor task(s), in terms of extra computational cycles. Hence, the additional cycles (IE_i) of a task (T_i), due to propagated errors by its predecessor(s) can be defined as:

$$IE_i = \sum_{p=1}^{PAR} OE_p \tag{3}$$

where PAR is the number of predecessors of T_i. This propagated error extends the mandatory part of the successor (T_j), since more processor-cycles are needed by the task to process the error. The extended mandatory part of a task (T_j) can be expressed as:

$$EM_j = M_j + IE_i \tag{4}$$

C. Generating Schedule and V/F Readjustment

The scheduling process of ARCTIC-*Offline* is given in Algorithm 1. The task graph ($G(T, E)$), in addition with the individual task's versions (k_i), execution lengths (len_j^i) and deadline (D_{PTG}) are the inputs to this algorithm. The algorithm also considers Acc_i^j and F_l^j as inputs, which are the accuracy achieved by executing the jth version of T_i at a frequency level of L for processor j. With the onset of the schedule generation process, Algorithm 1 employs the ILP based scheduling discussed in Sec. IV-A (line 2). The ILP considers power, precedence and time constraints and available cores to generate the primary schedule, $PrimarySched$. The generated schedule might not be able to execute all the tasks with their highest versions. By considering that schedule, OE_i for individual T_i is generated next and subsequently IE_i will be derived by employing Equation 2 and 3, before deriving EM_j by employing Equation 4. To generate EM_j, the list of predecessor for all tasks are also considered. The entire process of generating EM_j is written in line 3 to 7.

Once the extended mandatory part is generated (EM_j), the $PrimarySched$ is next modified and this schedule is named $SchedMod$ (line 8). As $SchedMod$ is burdened with additional execution cycles, this might result in a violated deadline. Any task T_η, the last task (can potentially be the sink task) that has been scheduled on processor P_j ($T_\eta^{P_j}$) and having its finish time after the deadline, i.e. $FinishTime(T_\eta^{P_j}) > D_{PTG}$, trivially implies that the deadline is violated (line 10). Upon detection of such deadline violation, Algorithm 2 will be called (line 11), where existing possible idle slots at some cores will be exploited as much as possible to adjust the frequency of the non-idle cores by respecting power constraints.

To update the $SchedMod$, Algorithm 2 first considers the tasks scheduled at each processor (P_i). Now, for each P_i, Algorithm 2 detects the idle slots by considering finish time and start time of two subsequent tasks (line 1 to 5). For each such idle slots at a particular processor (P_i), Algorithm 2 searches if any other processor(s) (P_j) is scheduled with a lower frequency, (where $P_i \neq P_j$), then the frequency of P_j is set at the highest level for that slot (line 6 to 8). Such overlap slots between a pair of processor cores are detected by listing the idle slots of each individual processors ($IdleSlots(P_i)[]$). For such individual idle slots of a particular processor (P_i), all other processors' schedules will be verified and the active-idle overlap pairs will be detected accordingly. However, such changes in frequency will reduce the execution lengths of the individual tasks, for which $SchedMod$ will be updated and the updated schedule is named as $SchedUpdate$, which is next returned to Algorithm 1 (line 9 to 10 in Algorithm 2). Upon receiving the updated schedule, Algorithm 1 checks if this new schedule is meeting the deadline (line 12). If a new schedule meets its deadline, Algorithm 1 stores this final schedule $SchedFinal$ in dispatch table and returns the QoS (line 17). Otherwise, the schedule will be discarded (line 15).

D. Example: Constrained scheduling at work

By applying the ILP based mechanism for the proposed scheduling problem based on the PTG depicted in Figure 2,

Algorithm 2: Exploit_Idle_Slots ($SchedMod$)

1 **for** *each processor* (P_i) **do**
2 List the tasks scheduled at P_i by considering $Sched_Mod$;
3 **for** *each pair of subsequent tasks* (T_i, T_j) **do**
4 **if** $StartTime(T_j) > FinishTime(T_i)$ **then**
5 $IdleSlots(P_i)[] = \{StartTime(T_j) - FinishTime(T_i)\}$;
6 **for** *each entry in* $IdleSlots(P_i)[]$ **do**
7 **if** *P_j is scheduled with lower frequency, where $P_i \neq P_j$* **then**
8 Set the frequency (F_l^j) at the highest level for P_j ;
9 Re-calculate the time-stamps and spans for each task in $SchedMod$ along with its successors, for which frequency has been readjusted, and the new schedule is named as $SchedUpdate$;
10 Return $SchedUpdate$;

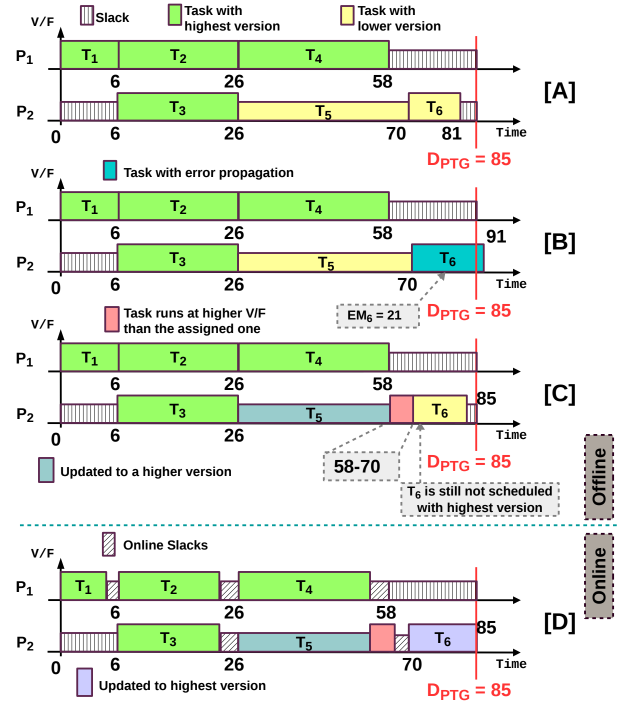


Fig. 3: Generated Schedule and V/F readjustment, and online LLC induced improvement (not to scale).

Algorithm 1 at first generates the $PrimarySched$ as shown in diagram [A] of Figure 3, whereas our assumed system has two processors (P_1 and P_2) and with an overall power-budget of 35 units.

Note that, the source task does not have any such overhead. In Figure 3, T_5 is scheduled with a lower version, hence T_6 will be scheduled with an extended mandatory part EM_6 as shown in diagram [B], which is the modified schedule, $SchedMod$, derived by Algorithm 2. Such error propagation leads to deadline violation for $SchedMod$, which is shown in diagram [B] of Figure 3.

The assigned V/F levels of a processor is adjusted by observing the assigned V/F level of the other processor(s) so that the power constraint is not violated. During task-allocation, the V/F level for each task on a specific processor is determined. For a pair of processors, if there exist any overlapped time-slots where one processor is idle, and the other one is executing a task at some lower V/F, then ARCTIC

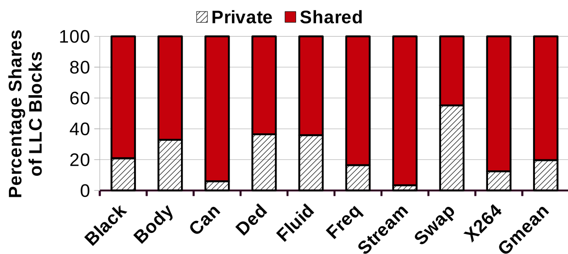


Fig. 4: Percentage shares of Private and Shared LLC Blocks.

will increase V/F of that processor (shown in Figure 3[C]). Note that, this task-level V/F readjustment will reduce the execution length of tasks, and thus will improve the QoS. In Figure 3[C], our $SchedFinal$ is depicted, in which the V/F adjustment reduces the execution length of the task T_5 in diagram [B]. Hence, completion time of T_5 remains same at 70, however, a higher version has been scheduled, thanks to V/F adjustment. Even after scheduling a higher version of T_5, T_6 can also be completed without a deadline-violation and our final obtained QoS is 46 (Figure 3[C]). The feasible schedule, $SchedFinal$ is next stored into the dispatch table [27] by Algorithm 1, and subsequently, the tasks will be executed.

V. ARCTIC-*Online*

In this section, we will illustrate the ARCTIC-*Online* mechanism after discussing the background on private blocks. Basically, the schedule generated by Algorithm 1 is deployed for execution and the online mechanism attempts to improve QoS and energy efficiency.

A. Analysing Exclusive Cache Blocks

To analyse the presence of shared and private blocks in the LLC, we executed a set of PARSEC benchmarks [9] in gem5 [10]. The footprints of private and shared blocks have been captured once the execution of an application proceeds to the region of interest (ROI). The traces have been captured at the end of each million of clock cycles, and we took this traces for 80 million clock cycles within the ROI[1]. The percentages of private and shared LLC blocks are shown in Figure 4. Our analysis shows, on average, about 20% of all LLC blocks are private, and can be as high as 58% for some applications (e.g., $Swap$). As we consider an inclusive cache hierarchy, a significant LLC-space is occupied by the private blocks. However, a single copy of such exclusive cache blocks in the respective requester cores' private caches can maintain the functional correctness and thus a large portion of the LLC will be freed up, which can be filled up by more live blocks to improve performance.

In case of an exclusive cache hierarchy, the effective cache capacity is significantly higher than the inclusive ones. But, maintaining coherency at the shared cache in exclusive cache hierarchy can incur significant costs in terms of latency and energy usages [46]. Implementation of coherence state machine for the exclusive cache hierarchy is more complex than its inclusive counterparts, and exclusive cache also requires an additional victim buffer, which will further incur extra area as well as energy overhead. While considering the sharing pattern between blocks as shown in Figure 4, it can be stated that, the larger percentage of the LLC blocks are shared in nature. Hence, implementing exclusive cache hierarchy can incur significant energy, area and coherence overheads [18], [19]. In ARCTIC-*Online*, we therefore propose an inclusive cache hierarchy with slight modification in the coherence management scheme, that will handle the private blocks like an exclusive cache, while shared blocks will be handled normally like the inclusive cache. Such technique will not incur energy, area, coherence or implementation overheads like exclusive cache, but can exploit the benefits of the exclusive cache for the private blocks.

[1]By evaluating LLC traces for longer time-span (of 200 million clock cycles), we observed that the shares of private and shared blocks does not change significantly. Hence, we decided to take the values derived at the 80 million clock cycles.

B. Core Concept of ARCTIC-Online

The entire mechanism of ARCTIC-*Online* is governed by Algorithm 3. This algorithm handles LLC management (described in Sec. V-B1) techniques along with the online QoS improvement and energy savings. Time span between the starting time till deadline is called $Frame$, within which the scheduled tasks need to be executed. The tasks are fetched from the dispatch table for execution (line 3), and while executing the tasks, the LLC management algorithm (Algorithm 4) is executed at each LLC bank simultaneously (line 5 to 7). On the other hand, the core based QoS improvement and energy savings are also handled at each core simultaneously (line 8 to 10) by employing Algorithm 8 (discussed in Sec. V-B2).

Algorithm 3: ARCTIC-*Online* Mechanism

```
1  for each Frame do
2      for all T_i in Dispatch Table do
3          Get schedule details of each T_i from the Dispatch Table;
4          Fetch T_i and start execution;
5          for each LLC bank do
6              Call Algorithm 4;
7              # Execute simultaneously at each bank;
8          for each Core do
9              Call Algorithm 8;
10             # Execute simultaneously at each core;
```

1) Managing LLC Blocks: To maintain a single copy of the private cache blocks in the inclusive cache hierarchy, the coherence protocol must be updated to ensure functional correctness. In ARCTIC-*Online*, the allocation will be performed in the following manner. We have segregated three cases for managing coherency in ARCTIC-*Online*, and the sequence of operations for each one is shown in Figure 5, whereas the whole implementation framework is given in Algorithm 4. In case **[A]** in Figure 5, once a miss is detected at both L1-D (1) and LLC (2) (line 3 in Algorithm 4), the request will be sent to the main memory (line 4) and an entry will be created in the directory (line 5). Upon receiving the memory response (line 11 to 12), the associated request type (read/write) and the number of requester(s) will be determined, the response management framework is detailed in Algorithm 7. When a miss is detected at the local data cache, but the data is

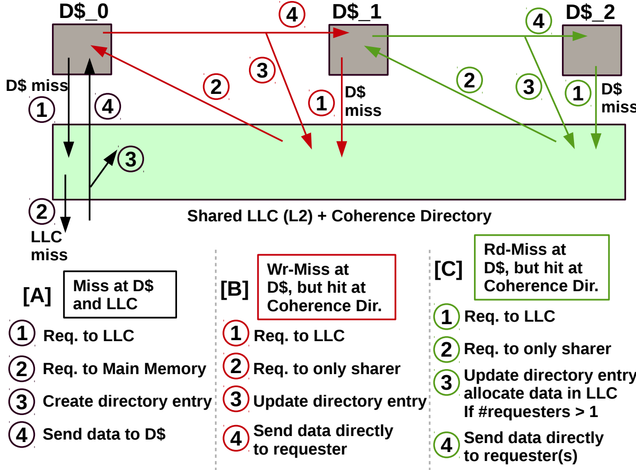


Fig. 5: Request-Response Paradigm across Cache Hierarchy.

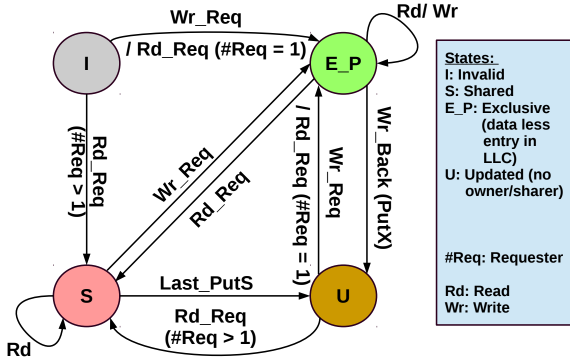


Fig. 6: LLC Coherence State-Transitions with events.

exclusively present in another data cache, the LLC will be requested at first for the data. In such cases (case **[B]** and **[C]** in Figure 5), there will be a hit in the coherence directory, and data will be forwarded to the new requester (line 7 to line 8), and Algorithm 5 will be called. However, to handle LLC hit, Algorithm 6 is called by Algorithm 4 (line 9 to 10).

Algorithm 4: LLC Management

```
1  ULC: Upper Level Cache, C_D: Coherence Directory, DLT: Data
     Less Tag Array, NT: Normal Tag Array
2  for each Request (Req) or Response (Resp) to LLC bank do
3      if Req.tag is missed in both DLT and NT then
4          # Req sent to next level of Memory;
5          # C_D.state = I;
6      else
7          if Req.tag hits in DLT then
8              Call Algorithm 5 ;
9          if Req.tag hits in NT then
10             Call Algorithm 6 ;
11         if Resp came from Main Memory then
12             Call Algorithm 7;
```

Handling Hit at Coherence Directory (Algorithm 5): Once there is a data L1 cache miss, the normal tag (NT) array is searched at the LLC along with the tag array maintained at the coherence directory (C_D). The tag array at the coherence directory is filled up with a tag with the exclusive copy of the data is present in any of the local data cache (defined as E_P coherence state), but no LLC copy is present in the LLC data array. This tag storage at the coherence directory is named Data Less Tag (DLT) array. However, upon receiving a request from a local cache, the request type needs to be evaluated first and the subsequent actions are taken accordingly (Algorithm 5). If it is a read request with more than one requester cores, then, C_D will forward the data directly to all new requester cores' data cache (as shown in (4) in Figure 5) and update the sharers' list at the C_D. Additionally, the data will be allocated in LLC after migrating the tag from DLT to NT. The entire process is given between line 1 to 6 in Algorithm 5. Case **[C]** in Figure 5 shows the same sequence of these operations, and the respective changes in the coherence state is shown in Figure 6. The coherence state transits from E_P to shared (S) state.

Upon receiving a write (exclusive) or a read request with one requester (line 7), i.e. case **[B]** in Figure 5, (4) the data will be sent to the requester core's private cache, after updating (3) an entry in the LLC coherence directory for the block. The directory entry contains the tag of the data, its coherence status and the requester core ID as an owner. The entire mechanism is given between line 7 to 10. Note that, as an exclusive copy of the data block will only move from one local cache to another, the coherence state will remain the same at E_P, which is shown by a self loop at this state in Figure 6.

In case of a write back request from the local cache in E_P state (line 12), the data will be allocated in the LLC, and the owner's entry will be cleared along with migrating the tag from DLT to NT (line 12 to 16). Finally, the coherence state will be changed to update state (U) and the respective transition is shown in Figure 6.

Algorithm 5: Handling DLT Hit

```
1  if Req.type == Rd_Req and #Req.ID > 1 then
2      # C_D forwards Req to Local owner cache and adds Req.ID
         to its sharer list;
3      # Local owner cache directly sends data to the local cache(s) of
         the requester(s) (Req);
4      # Migrate the DLT.tag from DLT to NT;
5      # Allocate data from the owner cache to LLC;
6      # C_D.state = S;
7  if Req.type == Wr_Req or (Req.type == Rd_Req &&
     #Req.ID == 1) then
8      # C_D forwards Req to Local owner cache and replaces owner
         entry with Req.ID;
9      # Local owner cache serves the Req and invalidates own copy;
10     # C_D.state remains E_P;
11 else
12     if Req.type == Wr_Back(PUTX) then
13         # Allocate updated data at the LLC from owner cache;
14         # C_D clears respective owner entry;
15         # Migrate the DLT.tag from DLT to NT;
16         # C_D.state = U;
```

Handling NT Hits (Algorithm 6): If the requested data block is found in the LLC, the hit will be declared at the NT array by following the conventional mechanism. Based upon the request type (line 1), the block is next handled by following the techniques given in Algorithm 6. If there is either a write request or a read request with one requester, then the block will be put in E_P state after invalidating its prior sharers and sending it to the requester (line 2 to 5). Note that, tag will also be migrated to DLT from NT array.

Algorithm 6: Handling NT Hit

```
1  if (Req.type == Wr_Req) or ((Req.type == Rd_Req) and
     (C_D.sharer == 1)) then
2      # Send Invalidation to all the sharer(s) and send block to the
         Req.ID;
3      # Reset Sharer list in C_D and initialize owner entry with
         Req.ID;
4      # Migrate the NT.tag from NT to DLT;
5      # C_D.state = E_P;
6  if Req.type == Last_PUTS then
7      # Reset Sharer list in C_D;
8      # C_D.state = U;
9  else
10     if (Req.type == Rd_Req) and (C_D.sharer > 1) then
11         # Serve the Req from NT;
12         # Add Req.ID in C_D sharer list;
13         # C_D.state remains S;
```

Algorithm 7: Handling Response from Main Memory

```
1  if (Resp.type == Wr_Req) or ((Resp.type == Rd_Req) and
     (#Req.ID == 1) then
2      # Allocate Data less entry in DLT;
3      # Send Resp.Data to requester local cache;
4      # Create C_D with owner entry set to requester local cache ID;
5      # C_D.state = E_P;
6  else
7      if (Resp.type == Rd_Req) and (#Req.ID > 1) then
8          # Allocate Resp.tag and Resp.Data in LLC NT;
9          # Serve the Resp.Data to requester local caches;
10         # Create C_D with sharers list initialized to requester
             cache IDs;
11         # C_D.state = S;
```

If a shared block is evicted from all of its sharers' L1s ($Last_PutS$), the state is changed to U from S (line 6 to 8). Upon receiving a read request for a shared block having more than one sharer (line 10), the request is served from NT with updating the sharers' list in the directory and the coherence state will remain the same at S (line 11 to 13). The respective transitions of the coherence states are depicted in Figure 6.

Handling Response from Memory (Algorithm 7): In case of an exclusive-read (having single requester) or a write LLC miss, the data is directly sent to the requester's data cache upon receiving the response from the main memory. The coherence state of the data is directly made as E_P by creating an entry in the DLT with updating the owner of the block (line 1 to 5). In case of a read LLC miss, while having multiple requester cores, the tag and data are allocated in the LLC's data array and NT array, and the copies of the data will be sent to the requester cores' local cache after updating the sharers' list in the coherence directory and setting the coherence state as S (line 7 to 11). The detailed coherence state transitions are given in Figure 6.

2) Improving QoS & Energy Efficiency (Algorithm 8): By maintaining exclusive cache entries for the private blocks, ARCTIC-*Online* frees up empty LLC spaces, that are further used to increase live block count. Such mechanism minimises LLC misses, which improves performance of the tasks and reduces the execution time of the individual tasks. After executing the mandatory portion (M_i) of the individual tasks, ARCTIC-*Online* evaluates its remaining time for executing the task (line 1 to 3). To calculate remaining execution

Algorithm 8: Improving QoS and Energy Efficiency

```
   Input: Break_Even_Time
1  # Execute M_i as per schedule ;
2  if Execution of M_i is over then
3      Cyc_rem_{O_i} = (Cyc_Ext_End_T_i − Cyc_End_M_i) ;
4  if Highest O_i is not scheduled then
5      # Call the function that returns optional part with highest
         possible accuracy which can run within Cyc_rem_{O_i} ;
6      O_i = get_O_i(T_i, Cyc_rem_{O_i}) ;
7      if O_i then
8          #Fetch the O_i ;
9  if Execution of T_i is over then
10     Slack_after_T_i = (Cyc_Ext_End_T_i − Cyc_End_T_i) ;
11 if Slack_after_T_i > Break_Even_Time then
12     #Power gate the core ;
13 if Core is power gated then
14     Slack_after_T_i--;
15     if Slack_after_T_i == Break_Even_Time then
16         #Turn on the core ;
```

time ($Cyc_rem_{O_i}$) for each task, we introduced extended end time of each task ($Cyc_Ext_End_T_i$), which is defined as follows. $Cyc_Ext_End_T_i$ for T_i is set as the starting time either of the next task at the same core or of its successor task, whichever is earlier. If T_i is a sink task, then $Cyc_Ext_End_T_i$ is set at D_{PTG}. However, $Cyc_rem_{O_i}$ is calculated by considering the time-span between the current time-stamp (at the end of M_i) and $Cyc_Ext_End_T_i$. If the highest version of the task is not scheduled, $Cyc_rem_{O_i}$ is used to decide if a higher version of the task can be executed, subject to availability (line 4 to 6). Otherwise, the task version will remain unchanged, i.e. the scheduled O_i will be executed (line 7 to 8). This will assist in improving overall QoS during task execution. However, as execution time of the O_i will also be minimised due to cache based performance improvement of ARCTIC-*Online*, there is a chance of generating dynamic slack after execution of each task. If such slack exists and its span (evaluated at line 11 to 12) is more than break even time of the respective processor core, the core will be shutdown for energy saving, and will be turned on before starting-time of the next task (line 13 to 16).

3) ARCTIC: Online Computational Overhead:

Theorem 1. *The amortized complexity of ARCTIC-Online (Algorithm 3 to 8) is* $\frac{\mathcal{O}(n \cdot k)}{D_{PTG}}$ *per time-slot.*

Proof. Algorithm 3 is the master algorithm of ARCTIC-*Online* technique that executes tasks at each core. A step-wise analysis of computational overhead of Algorithm 3 due to the called functions/algorithms is as follows:

1) The "for loop" from line 2 to 10 may be executed $\mathcal{O}(n)$ times in the worst-case, although the number of tasks assigned to a core usually takes a small value.
2) Next, the "for loop" from line 5 to 7 will be executed in constant time, as the number of LLC bank is always constant.
    - Algorithm 4 to 7 are called next during task execution. For all practical purposes, computational overheads of these algorithms may be considered to be constant, however, implementation overheads for Algorithm 5 and 7 is discussed in Sec. V-B4.

3) The "for loop" from line 8 to 10 will be executed in constant time, as the core count is always constant.
   - In worst-case, for Algorithm 8, the loop will execute line 4 to 8 which can have a worst-case complexity of $\mathcal{O}(k)$, where k is the maximum number of versions for a task T_i.
4) Hence, the worst-case computational complexity of Algorithm 3 is $\mathcal{O}(n \cdot k)$.
5) At any $FRAME$, the total overhead for generating the schedules over all processor cores for the duration of a $FRAME$ is $\mathcal{O}(n \cdot k)$ in the worst case.
6) As the $FRAME$ length is in $\mathcal{O}(D_{PTG})$, the amortized complexity of ARCTIC*: Online* is $\frac{\mathcal{O}(n \cdot k)}{\mathcal{O}(D_{PTG})}$.

□

4) Implementation Overhead: The implementation of ARCTIC-*Online* requires additional tag entries at the LLC, called DLT, for bookkeeping the tag and coherence states of the private blocks. Each LLC tag entry will be equipped with DLT space, which will be filled up with the tag entries of the blocks having the same index. We analysed the power and area overheads for adding 25% and 50% extra tags for implementing DLT and corresponding area and power overheads have been analysed with FinCanon [25] by considering a 14 nm FinFET technology. With 25% (50%) extra tag, LLC will experience an area overhead of 2.32% (4.53%), with a power overhead of 2.71% (5.64%). The power and area overheads are not significant, due to lower leakage consumption and higher cell-density of the FinFET-based caches [25]. To implement Algorithm 8, the per-core power gating (PCPG) technique can be employed at the circuit level, which does not incur any significant circuit overhead [26].

VI. Evaluation

In this section, we first showcase the efficacy of ARCTIC-*Offline*, followed by the evaluation of ARCTIC-*Online*, and show how ARCTIC improves system level QoS.

A. Methodology

1) ARCTIC-Offline: We define **N**ormalized **O**btained **Q**oS (NOQ) as the ratio between the actually achieved QoS for the PTG, and the maximum achievable QoS by executing the highest versions of all tasks. NOQ can be formulated as: $NOQ = \frac{\sum_{i=1}^{n} Acc_i^j}{\sum_{i=1}^{n} Acc_i^{k_i}}$, where k_i represents the highest version of task T_i. Next, we model our multi-core and the task-set:

- *Processor System:* A homogeneous multi-core platform having 4 Intel $x86$ cores (i.e., $m = 4$) has been considered. The TDP of the individual cores are scaled and set as 10.5W, by considering the Intel Xeon's data-sheet [1]. The runtime core power is obtained through McPAT [28].
- *Task-set:* Task characteristics have been taken from a prior technique, *Prepare* [12], where tasks are framed by using PARSEC benchmark applications. The total execution requirement of a PTG (C_{PTG}) is the sum of the execution times of its subtasks, $C_{PTG} = \sum_{i=1}^{n} ET_i$. Thus, utilization U_i of a PTG can be written as $\frac{C_{PTG}}{D_{PTG}}$.

TABLE I: System parameters [CC: clock cycle]

Parameter	Value	Parameter	Value
ISA	Intel x86	L1-I	64KB, 4Way, 3CC
#Cores (type)	8 ($Xeon$)	L1-D	64KB, 4Way, 3CC
Max. V/F (Max_VF)	1.12V, 3.0GHz	L2	8MB, 16Way, 12CC
Min. V/F (Min_VF)	0.6V, 1.5GHz	Cache	Non-MRU, 64B blocks
Power_gate_overhead	60 ns	DRAM latency	70 ns
ROB Size	200	Technology	14 nm FinFET
Dispatch/Issue width	8	Ambient Temp.	47 °C

The average utilization of a PTG is taken from a normal distribution, by considering a normalized frequency of 0.6. Given the PTG's utilization, we next obtain the total utilization of the system (Sys_{uti}) by summing up the utilization of all PTGs. Given the Sys_{uti}, the total system workload (Sys_{WL}) / system pressure can be obtained by: $Sys_{WL} = \frac{Sys_{uti}}{m}$. For a given Sys_{uti}, we generated all of our PTGs by following the method proposed in *Prepare* [12]. Given a Sys_{WL}, a set of PTGs are created. The number of PTGs (ρ) within a set is calculated as: $\rho = \frac{m \times Sys_{WL}}{U_i}$. In our generated PTGs, the minimum number of tasks is 5 and the maximum number of tasks is 20. For each PTG in the set, the number of tasks are generated randomly within a preset limit. Note that, as the individual U_i of a PTG is lower than the given Sys_{WL}, the number of PTGs (ρ) within the set will always be higher than m.

- *Task Temporal Parameters:* For each T_i, based on which portion of the len_i is considered as the mandatory portion (M_i), the following cases are considered [20]: *(i)* $man_low : M_i \sim U(0.2, 0.4) \times len_i$ (low portion of a task T_i's length (len_i) is for the mandatory portion). *(ii)* $man_med : M_i \sim U(0.4, 0.6) \times len_i$ (medium portion of a task T_i's length (len_i) is for the mandatory portion). *(iii)* $man_high : M_i \sim U(0.6, 0.8) \times len_i$ (high portion of a task T_i's length (len_i) is for the mandatory portion).

2) ARCTIC-Online: Simulation Infrastructure: In this work, we simulated a homogeneous CMP, having 8 $Intel$ $x86$ Xeon OoO cores in the gem5 full system simulator [10]. Each core is equipped with its private L1 data and instruction caches. The L2 cache, considered as LLC, is logically shared among the cores. The performance traces derived from gem5 are sent to McPAT-Monolithic [22], to generate the power traces, and system parameters used in the simulations by considering 14nm FinFET technology nodes are listed in Table I. In our simulation framework, each core can execute tasks either at Max_VF (3.0GHz) or at Min_VF (1.5GHz). By considering prior arts, where PARSEC [9] can be used in an approximate computing based paradigm [2], [41], we framed our task-set by defining each task with 4 PARSEC applications with large input sets. We constructed each M_i and O_i by using two copies of two different PARSEC applications, where the execution lengths of M_i and O_i for each task are set by scaling each values of M_i and O_i of Figure 2 with 20M clock cycles. Our multi-programmed task-set is detailed in Table II, where the execution lengths (EL) are given in millions of cycles in the region of interest (RoI) for the respective M_i's and O_i's. Each task's M_i and O_i run on a group of 4 cores, and we consider two such groups to represent P_i in Figure 3.

TABLE II: Tasks formation with PARSEC [9], [38]. *Black (2)* implies two copies of *Black*, which is the same for others. (Acronyms: Blackscholes (*Black*), Bodytrack (*Body*), Canneal (*Can*), Dedup (*Ded*), Fluidanimate (*Fluid*), Freqmine (*Freq*), Streamcluster (*Stream*), and X264 (*X264*)). The execution lengths (ELs) are in million cycles.

Tasks	Benchmarks (M_i, O_i)	EL ([M_i], [O_i])	Sel. Oi [EL]
T_1	*Black (2), Body (2)*	[80], [40]	#1 [40]
T_2	*Stream (2), Swap (2)*	[200], [100, 160, 200]	#2 [200]
T_3	*Ded (2), Can (2)*	[200], [100, 140, 200]	#3 [200]
T_4	*X264 (2), Fluid (2)*	[400], [140, 240]	#2 [240]
T_5	*Freq (2), Swap (2)*	[340], [100, 220, 300]	#2 [220]
T_6	*Ded (2), Body (2)*	[200], [20, 120]	#2 [20]

B. ARCTIC-*Offline*

ARCTIC-*Offline* is evaluated based on task rejection rate (TRR), which is the ratio of the number of tasks that did not finish their execution within their deadline, over the number of all tasks executed by the system. We obtain task characteristics and the total utilization of the system (Sys_{uti}) from a prior art, proposed by S. Saha et al. [38]. The left plot in Figure 7 depicts the TRR obtained by ARCTIC-*Offline* for 40, 60 and 80% of Sys_{WL}. We observed that TRR remains 8%, when the system workload is low, which is increased to 35% on average, when the workload is scaled up by 40%. As the system workload is increased to maintain the number of PTGs in the system, the individual PTG-utilization also increases, that eventually contributes to high TRR. Actually, increased utilization results in longer running time of each node, that trims the possibility of obtaining sufficient free slots within the deadline. Moreover, the insufficient free slots curtail the chances of selecting higher task versions, thus increase in the error propagation. This cascading effect increases the execution length of the sink node that violates the deadline.

In case of man_{high}, the *TRR* increases in a lower rate than for man_{med} and man_{low}, while increasing Sys_{WL} (shown in Figure 7 (Left)). Basically, when mandatory portions of the individual tasks are high, the length of the optional portions will be low, that results into lower variance among the different versions of a task, which reduces error propagation.

The right plot in Figure 7 shows the reduction in TRR ($RTRR$) for different values of Sys_{WL}. We compared the changes in $RTRR$ for ARCTIC-*Offline* (suffixed by _AR) with the scheduling proposed in *TD* [35] (suffixed by _TD). We observed that, increased Sys_{WL} trivially reduces $RTRR$ for ARCTIC-*Offline*, which still significantly outperforms *TD* for all the workloads. This is because *TD* accounted no power-budget, whereas the assumed energy budget increased with higher number of tasks. Moreover, *TD* also allows unlimited task migration that incurs extra overheads. For a given Sys_{WL}, man_{low} obtains a higher reduction in TRR, as the lengths of the mandatory (optional) parts of individual tasks are short (long), that also increases the probability of obtaining slacks.

We have compared our policy with a prior strategy ($SENAS$) [21] and the results are shown in Figure 8 in case of man_med. For a fair comparison with $SENAS$, we firstly derived the overall energy limit based on our considered power budget (Pow_{BGT}) of ARCTIC's experimental framework. The same value is used as energy budget for $SENAS$ as well. It can be observed, as the number of task increases (due to increase in Sys_{WL}), that ARCTIC maintains higher QoS by achieving higher NOQ than $SENAS$. ARCTIC is able to maintain close to 70% QoS with 70% workload, where $SENAS$ achieves 55% QoS. This is because $SENAS$ did not consider any power limits but assumed a fixed energy budget. Moreover, $SENAS$ assumed that the task's energy consumption increases with its execution cycle, i.e., the higher the task's execution, the higher the energy requirement. Hence, for a fixed energy budget, $SENAS$ fails to execute tasks with a higher version and thus, ends up with a lower QoS.

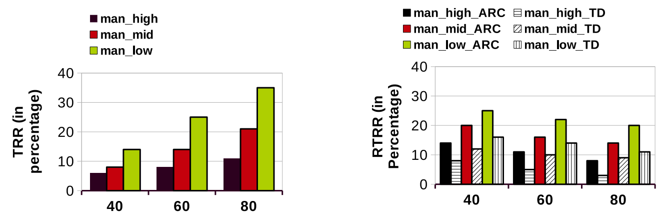


Fig. 7: System Workload (in percentage) vs. TRR (Left), and RTRR (Right).

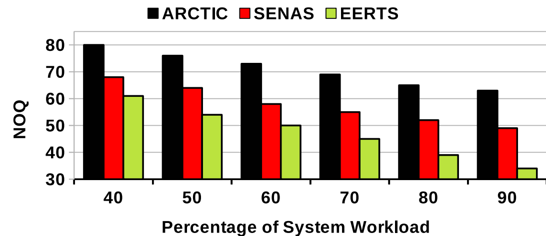


Fig. 8: Comparing NOQ: ARCTIC vs. Prior arts.

We compared our offline strategy also with EERTS [7] and it can be observed from the figure that EERTS achieved considerable NOQ with low system utilization. However, NOQ decreased significantly with the increase in system workload. This is mainly due to the fact that EERTS only considered tasks with a linear task chain and considered only the latency constraint but not the power budget has been considered. Now, when the system workload is low, the utilization of individual tasks is also low, and this correspondingly attributes to a lower number of tasks/nodes within a graph (Sec. VI-A). Thus EERTS achieved close to 60% NOQ. However, as the workload increases, the number of nodes also increases, making the graph structure more skewed. Hence, EERTS fails to achieve a higher QoS, as tasks with a higher versions cannot be scheduled due to stringent power, resource and deadline constraints.

C. ARCTIC-*Online*

1) Performance Improvement and EDP gains: We first evaluate the reduction in LLC misses (MPKI), and the IPC values for each task, with two different extra DLT storage of 25% (ARC_25) and 50% (ARC_50). We also compare these results with a prior LLC *way-sharing and way-gating* based technique, *ACCURATE* [38] (ACCRT), the results for MPKI reductions are shown in Figure 9 for all tasks. For T_3

and T_4, the higher number of shared LLC blocks curtails the benefits of ARCTIC-*Online*, whereas for T_2, T_5 and T_6 higher private block counts lead to significant reduction in MPKI. The MPKI reduction is higher in ARC_50 than ARC_25, due to accumulation of more live blocks. For tasks like T_3, having higher private block count during M_i and higher shared blocks during O_i, the performance improves further during M_i than during O_i, thus the slack-span is getting longer. On the other hand, T_4 contains higher shared block counts during M_i, which lowers the performance benefits during M_i. For all tasks, ACCRT achieves lower reduction in MPKI (9.4% on average) than both ARC_25 (13.5% on average) and ARC_50 (30.2% on average), as ACCRT turns off some portions of the LLC to reduce LLC leakage. However, this lower MPKI reduction leads to lower IPC improvement (2.7% on average) for ACCRT whereas ARC_25 and ARC_50 achieve 5.2% and 8.9% average improvements in IPC, respectively. Figure 10 shows the IPC improvements for all of our tasks for ARCTIC-*Online* and ACCRT. The higher the number of private blocks during both M_i and O_i in case of T_6 the more significant reduction in IPC, that enables to enhance QoS by executing higher version of O_i. For all tasks, ARC_50 reduces more MPKI during M_i than ARC_25, which generates longer slack in ARC_50, which in turn leads to higher energy savings. For individual tasks, we evaluated the EDP gains for ARCTIC-*Online* and ACCRT, as shown in Figure 11. The average EDP gains for both ARC_25 and ARC_50 are around 5.3% and 9.1%, respectively, that surpass the EDP gains of 2.7% for ACCRT. ARCTIC-*Online* power gates the cores for a sufficiently longer time-span during its longer slacks, that in turn leads to higher EDP gain.

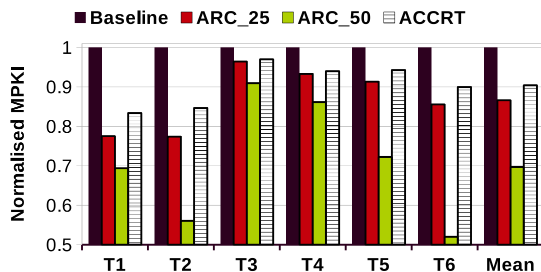


Fig. 9: Normalised MPKI.

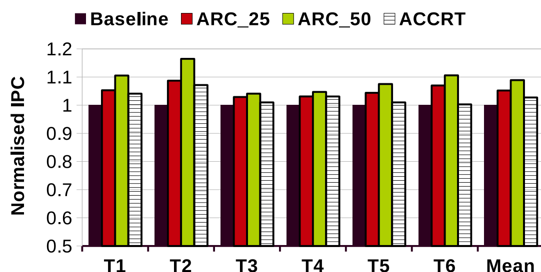


Fig. 10: Normalised IPC.

Figure 12 shows how ARCTIC-*Online* improves the QoS and energy efficiency by reducing execution length of the tasks, while applying ARC_50. ARCTIC-*Online* reduces the execution lengths of individual tasks, that generates slacks at the end of each task, during which cores are power gated to save energy. For all but sink tasks, starting time-stamps are kept the same, as derived by ARCTIC-*Offline*. However, once all predecessor tasks are completed, ARCTIC-*Online* schedules the sink (T_6) and based on the available time-span and the highest possible version of O_i is executed. In ARC_50, ARCTIC-*Online* executed the highest version of T_6, that improves QoS by 10%, with an overall energy saving of 6.21%, even after exploiting the slack. However, in ARC_25, the online QoS improvement is 4% with 12.32% energy saving, as T_6 is able to execute its second version within deadline. However, higher IPC improvements for all tasks in ARC_50 generate longer slacks, which are exploited to save more energy.

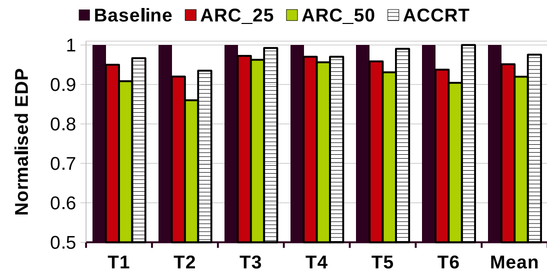


Fig. 11: Normalised EDP.

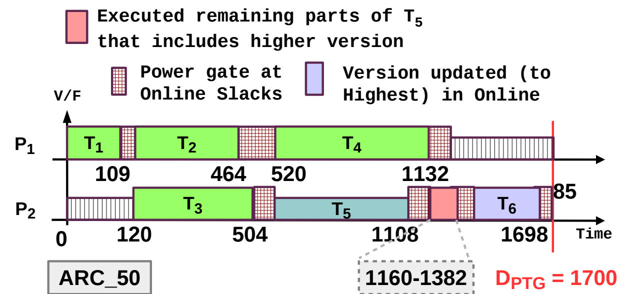


Fig. 12: QoS Update in Online for ARC_50. Timeline is based on Table II.

TABLE III: Outputs of ARCTIC-Online

Tasks	Mapped Core	Final Version	Amount of Slack (over Offline)
T_1	P_1	1	9.2%
T_2	P_1	3	14.0%
T_3	P_2	3	3.8%
T_4	P_1	2	4.4%
T_5	P_2	2	6.9%
T_6	P_2	2	9.6%
Improvement in Achieved QoS		10.0%	

D. Discussion: Applicability of ARCTIC in Heterogeneous Platform

To showcase the efficacy of ARCTIC-*Online* while employed in heterogeneous platforms, we further evaluated both ARC_25 and ARC_50 of ARCTIC-*Online* in our simulation

TABLE IV: Heterogenous System parameters [CC: clock cycle]

Parameters		Values
Big Core	ISA	RISCV (RV64GC)
	Cores	4 cores, single issue, in-order
	L1 Cache	4KB L1I/D, 2-way, 1 CC
Tiny Core	ISA	RISCV (RV64GC)
	Cores	60 cores, 4-way out of order 16 entry LSQ, 128 entry ROB
	L1 Cache	64KB L1I/D, 2-way, 1 CC
L2 Cache		Shared, 8-banks, 1MB per bank, 8-way, 1 bank per mesh coloumn
Network		8x8 mesh, XY routing, 16B per flit
Benchmark Suite: Ligra [40]		BC, BF, BFS, BFS-BV, CC, MIS, RADII and TC

infrastructure. We model heterogeneous multi-core system in gem5 [10], with different core types, e.g. Big-core and Tiny-core, employing the setup proposed by Wang et al. [44]. Table IV details the system parameters used for our heterogeneous simulation. In the simulated system, each core has its own private L1 data and instruction caches and all cores share the L2 cache, which is considered as LLC. Due to existing limitations related to support-ability of PARSEC benchmark applications in heterogeneous simulation infrastructure of gem5, we used Ligra benchmark suite [40], which is also used to construct task-sets in approximate computing paradigm [43]. The applications chosen for evaluation are also listed in Table IV.

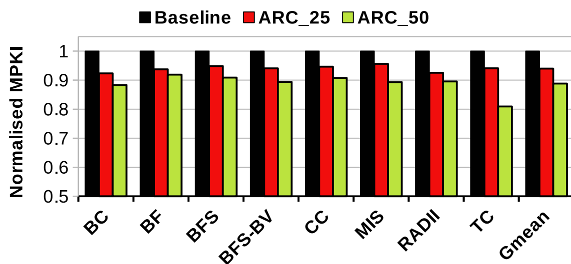


Fig. 13: Normalised MPKI: in Heterogeneous Multicore.

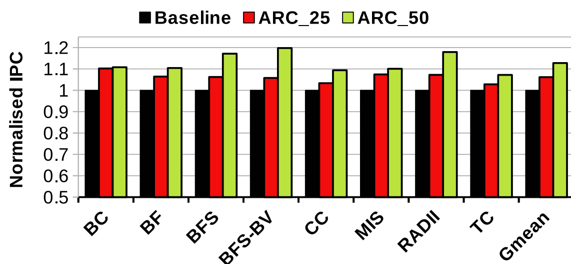


Fig. 14: Normalised IPC: in Heterogeneous Multicore.

Our evaluation shows, for all applications, ARC_50 shows higher reduction in MPKI than ARC_25, as we have seen in case of homogeneous system. The MPKI reduction for all applications are plotted in Figure 13. Overall, MPKI across all applications is reduced upto 8% and 12% for ARC_25 and ARC_50, respectively, over the baseline, whereas the respective average values for ARC_25 and ARC_50 are 6% and 11%. This reduction in MPKI further leads to performance improvement, which is shown in Figure 14, by plotting results for IPC. For all applications, the average IPC improvement is 6% and 13% for ARC_25 and ARC_50, respectively, whereas the respective ranges for IPC improvements for ARC_25 and ARC_50 are in 2-10% and 7-20%. This significant reduction in MPKI and IPC improvement in case of heterogeneous system demonstrates that ARCTIC can also be a promising technique in heterogeneous systems to stimulate QoS of an approximate real-time computing paradigm.

VII. Conclusions

In ARCTIC, we introduce a novel hybrid offline-online *scheduling strategy* for approximate real-time tasks. ARCTIC generates a schedule for a dependent task-set with an objective to maximise the QoS while respecting other system-wide constraints. ARCTIC-*Offline* further considers the tasks scheduled with lower accuracy and schedules the portions of their respective optional parts with their respective successors, known as Error Propagation. As Error Propagation might lead to deadline violation, ARCTIC-*Offline* also introduced a V/F readjustment technique at the task level granularity, while respecting the power budget and deadline. At runtime, ARCTIC-*Online* keeps the only copies of the private blocks in the respective local caches, and frees up the LLC locations. These spare LLC spaces are further utilized to boost performance by accumulating more live blocks on-chip, reducing tasks' execution-times. The slacks generated by reduced execution-time will further be exploited to enhance QoS by executing more from the tasks' optional parts or improve energy efficiency by power gating the core. Simulation results show that, for a set of tasks ARCTIC-*Offline* reduces task rejection rate up to 25%, whereas ARCTIC-*Online* yet improves QoS by 10% with 9.1% average EDP gain, while surpassing a state-of-the-art technique.

Acknowledgments

This work is supported by the UK Engineering and Physical Sciences Research Council (EPSRC) through grants $EP/X015955/1$, $EP/V000462/1$, and is also funded by *Marie Curie Individual Fellowship (MSCA-IF), EU (Grant Number* 898296*).*

For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) license to any Author Accepted Manuscript version arising.

References

[1] "12th generation intel® core™ processor family," https://www.intel.com/content/www/us/en/products/docs/processors/core/core-technical-resources.html, accessed: 2022-03-28.
[2] S. Achour and M. C. Rinard, "Approximate computation with outlier detection in topaz," *SIGPLAN Not.*, 2015.
[3] S. Agarwal and H. K. Kapoor, "Reuse-distance-aware write-intensity prediction of dataless entries for energy-efficient hybrid caches," *IEEE TVLSI*, 2018.
[4] J. Albericio *et al.*, "The reuse cache: Downsizing the shared last-level cache," in *MICRO*, 2013.
[5] M. Ansari *et al.*, "Peak-power-aware energy management for periodic real-time applications," *IEEE TCAD*, vol. 39, no. 4, 2020.

[6] H. Aydin *et al.*, "Optimal reward-based scheduling for periodic real-time tasks," *IEEE TC*, 2001.
[7] K. M. Barijough *et al.*, "Exploiting approximations in real-time scheduling," in *Approximate Computing Techniques: From Component-to Application-Level*. Springer, 2022.
[8] A. Bhuiyan *et al.*, "Energy-efficient parallel real-time scheduling on clustered multi-core," *IEEE TPDS*, 2020.
[9] C. Bienia *et al.*, "The PARSEC benchmark suite: Characterization and architectural implications," in *PACT*, 2008.
[10] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH CAN*, 2011.
[11] K. Cao *et al.*, "QoS-adaptive approximate real-time computation for mobility-aware IoT lifetime optimization," *IEEE TCAD*, 2019.
[12] S. Chakraborty *et al.*, "Prepare: Power-Aware Approximate Real-Time Task Scheduling for Energy-Adaptive QoS Maximization," *ACM TECS*, 2021.
[13] S. Chakraborty and H. K. Kapoor, "Analysing the role of last level caches in controlling chip temperature," *IEEE Transactions on Sustainable Computing*, vol. 3, no. 4, 2018.
[14] S. Chakraborty and H. K. Kapoor, "Exploring the role of large centralised caches in thermal efficient chip design," *ACM TODAES*, 2019.
[15] S. Chakraborty and M. Själander, "WaFFLe: Gated cache-ways with per-core fine-grained DVFS for reduced on-chip temperature and leakage consumption," *ACM TACO*, 2021.
[16] S. Chang *et al.*, "Toward minimum WCRT bound for DAG tasks under prioritized list scheduling algorithms," *IEEE TCAD*, 2022.
[17] J.-J. Chen *et al.*, "Scheduling of real-time tasks with multiple critical sections in multiprocessor systems," *IEEE TC*, 2020.
[18] H. Cheng *et al.*, "LAP: Loop-Block Aware Inclusion Properties for Energy-Efficient Asymmetric Last Level Caches," in *ISCA*, 2016.
[19] L. B. Drault, "Evaluation of cache inclusion policies in cache management," Master's thesis, Texas A & M University, August 2017. [Online]. Available: https://hdl.handle.net/1969.1/166081
[20] A. Esmaili *et al.*, "Energy-aware scheduling of task graphs with imprecise computations and end-to-end deadlines," *ACM TODAES*, 2019.
[21] K. Guha *et al.*, "SENAS: security driven energy aware scheduler for real time approximate computing tasks on multi-processor systems," in *IOLTS*, 2022.
[22] A. Guler and N. K. Jha, "McPAT-Monolithic: An area/power/timing architecture modeling framework for 3-D hybrid monolithic multicore systems," *IEEE TVLSI*, 2020.
[23] J. Huang *et al.*, "Dynamic DAG scheduling on multiprocessor systems: reliability, energy, and makespan," *IEEE TCAD*, 2020.
[24] K. H. Kim, "Reward-based allocation of cluster and grid resources for imprecise computation model-based applications," *International Journal of Web and Grid Services*, vol. 9, no. 2, pp. 146–171, 2013.
[25] C. Lee and N. K. Jha, "FinCANON: A PVT-Aware Integrated Delay and Power Modeling Framework for FinFET-Based Caches and On-Chip Networks," *IEEE TVLSI*, 2014.
[26] J. Lee and N. S. Kim, "Analyzing potential throughput improvement of power- and thermal-constrained multicore processors by exploiting DVFS and PCPG," *IEEE TVLSI*, 2012.
[27] K. Lee, *et al.*, "Mixed harmonic runnable scheduling for automotive software on multi-core processors," *International Journal of Automotive Technology*, 2018.
[28] S. Li *et al.*, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009.
[29] X. Li *et al.*, "Approximation-aware task deployment on heterogeneous multi-core platforms with dvfs," *IEEE TCAD*, 2022.
[30] M. Lodde *et al.*, "Dynamic last-level cache allocation to reduce area and power overhead in directory coherence protocols," in *Euro-Par Parallel Processing*, 2012.
[31] S. S. Manohar *et al.*, "Towards optimizing refresh energy in embedded-DRAM caches using private blocks," in *GLS-VLSI*, 2019.
[32] R. Medina *et al.*, "Generalized mixed-criticality static scheduling for periodic directed acyclic graphs on multi-core processors," *IEEE TC*, 2020.
[33] S. Mittal, "A survey of techniques for approximate computing," *ACM Comput. Surv.*, vol. 48, no. 4, 2016.
[34] ——, "A survey of techniques for cache locking," *ACM TODAES*, 2016.
[35] L. Mo *et al.*, "Approximation-aware task deployment on asymmetric multicore processors," in *DATE*, 2019.
[36] ——, "Energy efficient, real-time and reliable task deployment on noc-based multicores with dvfs," in *DATE*, 2022.
[37] M. Powell *et al.*, "Gated-Vdd: A circuit technique to reduce leakage in deep-submicron cache memories," in *ISLPED*, 2000.
[38] S. Saha *et al.*, "ACCURATE: Accuracy maximization for real-time multi-core systems with energy efficient way-sharing caches," *IEEE TCAD*, 2022.
[39] W. Shaogang *et al.*, "Optimizing private memory performance by dynamically deactivating cache coherence," in *HPCC*, 2012.
[40] J. Shun and G. E. Blelloch, "Ligra: A lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2013, p. 135–146.
[41] S. Sidiroglou-Douskos *et al.*, "Managing performance vs. accuracy trade-offs with loop perforation," in *ACM SIGSOFT*, 2011.
[42] Y.-C. Tian and D. C. Levy, *Handbook of real-time computing*. Springer Nature, 2022.
[43] K. Tovletoglou *et al.*, "HaRMony: Heterogeneous-Reliability Memory and QoS-Aware Energy Management on Virtualized Servers," in *ASPLOS*. ACM, 2020.
[44] M. Wang *et al.*, "Efficiently supporting dynamic task parallelism on heterogeneous cache-coherent systems," in *ISCA*, 2020.
[45] S. Yi *et al.*, "EASYR: E nergy-Efficient A daptive Sy stem R econfiguration for Dynamic Deadlines in Autonomous Driving on Multicore Processors," *ACM TECS*, 2023.
[46] Y. Zheng *et al.*, "Performance evaluation of exclusive cache hierarchies," in *ISPASS*, 2004.
[47] J. Zhou *et al.*, "Energy-adaptive scheduling of imprecise computation tasks for QoS optimization in real-time MPSoC systems," in *DATE*, 2017.

Sangeet Saha is currently associated with the Embedded and Intelligent Systems (EIS) Research Group, University of Essex, UK as a Lecturer. Prior to that, he worked as a lecturer at the University of Huddersfield, UK, and Senior research officer (Postdoctoral scholar) at the University of Essex, UK. His current research interests include real-time scheduling, scheduling for reconfigurable computers, real-time and fault-tolerant embedded systems, and cloud computing. He published several of his research contributions in conferences like CODES+ISSS, ISCAS, Euromicro DSD, and in journals like ACM TECS, IEEE TCAD, IEEE TSMC.

Shounak Chakraborty (Senior member, IEEE) is currently working as a guest researcher at the Department of Computer Science, NTNU, Norway. Primarily, his research interests include high performance computer architectures, emerging memory technologies, on-chip thermal management, and compilers. Prior to joining NTNU, Shounak obtained his PhD degree in Computer Science and Engineering from IIT Guwahati, India in February 2018, and also worked as assistant professor at IIIT Guwahati, India.

Sukarn Agarwal is an Assistant Professor at School of Computing and Electrical Engineering, IIT Mandi (India). He has earned his PhD degree in Computer Science and Engineering from IIT Guwahati, India, in March 2020, and was also a senior research fellow at School of Informatics, University of Edinburgh (UK). His research interests include Emerging Memory Technologies, Memory System Design, Network-on-Chip design and Thermal Aware Chip Management. He published many of his research contributions in conferences like IPDPS, DAC, PLDI, ASAP, VLSI-SoC, GLS-VLSI, etc. and also published several of his research outcomes in journals like IEEE TVLSI, ACM TECS, IEEE TC, and ACM TODAES.

Magnus Själander is working as a Professor at the Norwegian University of Science and Technology (NTNU). He obtained his Ph.D. from Chalmers University of Technology in 2008. Before joining NTNU in 2016 he has been a researcher at Chalmers University of Technology, Florida State University, and Uppsala University. Själander's research interests include hardware/software co-design (compiler, architecture, and hardware implementation) for high-efficiency computing.

Klaus McDonald-Maier is currently the Head of the Embedded and Intelligent Systems Laboratory and Director Research, University of Essex, Colchester, U.K. He is also the founder of UltraSoC Technologies Ltd., the CEO of Metrarc Ltd., and a Visiting Professor with the University of Kent. His current research interests include embedded systems and system-on-chip design, security, development support and technology, parallel and energy-efficient architectures, computer vision, data analytics, and the application of soft computing and image processing techniques for real-world problems. He is a member of VDE and a Fellow of the BCS and IET.