World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:14, No:11, 2020

A Study of the Trade-off Energy Consumption-Performance-Schedulability for DVFS Multicore Systems

Jalil Boudjadar

Abstract—Dynamic Voltage and Frequency Scaling (DVFS) multicore platforms are promising execution platforms that enable high computational performance, less energy consumption and flexibility in scheduling the system processes. However, the resulting interleaving and memory interference together with per-core frequency tuning make real-time guarantees hard to be delivered. Besides, energy consumption represents a strong constraint for the deployment of such systems on energy-limited settings. Identifying the system configurations that would achieve a high performance and consume less energy while guaranteeing the system schedulability is a complex task in the design of modern embedded systems. This work studies the trade-off between energy consumption, cores utilization and memory bottleneck and their impact on the schedulability of DVFS multicore time-critical systems with a hierarchy of shared memories. We build a model-based framework using Parametrized Timed Automata of UPPAAL to analyze the mutual impact of performance, energy consumption and schedulability of DVFS multicore systems, and demonstrate the trade-off on an actual case study.

Keywords—Time-critical systems, multicore systems, schedulability analysis, performance, memory interference, energy consumption.

I. INTRODUCTION

SINCE the introduction of Dual-core Itanium as a first multicore platform model by Intel early 2000's, multicore platforms are being the mostly used architecture for the deployment of real-time systems. Such platforms enable to leverage the computing capabilities, reduce the weight of on-board computing equipment and lower the energy consumption.

Multicore platforms either cannot be used in safety-critical systems (SCS) due to non-determinism and unpredictable memory interference, or if used the resources that they provide would be severely under utilized in order to ensure the system safety [26], [19], [3], [21]. In either case, one could deploy SCS on multicore platforms without under-utilizing their resources if absolute-guarantee analysis can be carried out to secure the safety. The most promising analysis methodology is model-based technology where formal methods can be applied [20], [7], [9]. Such a methodology drastically reduces the testing effort to be carried out when the final SCS is built.

Multicore platforms can experience a drastic energy consumption that is not always worth to pay for, in particular when the workload is not heavy or the timing constraints are not tight. e.g, during cruise mode an aircraft can run less workload compared to taking-off mode whereas during taxing mode some of the time constraints are usually less tighter, thus making the platform performance and consumption flexible is an asset.

The use of DVFS enables to save energy by tuning the cores frequency according to, among others, the workload and hardware temperature [29]. For real-time applications, systems supporting DVFS need to balance the achieved energy savings with the time constraints of applications. However, DVFS adds a third dimension to the design, complexity and certification of safety critical systems. The use of DVFS may lead to different non-deterministic runtime scenarios [32]. The processing cores of a DVFS platform can be independent and run individual workloads, for example the ARINC partitioning for Integrated Modular Avionics (IMA) architectures [31], where the frequency is adjusted on a per-core basis [25] making the execution of a task shorter (respectively longer) may lead to violate the data flow predictability and synchronization between tasks [8]. For example, when a processing task *A* is allocated to a core running a higher frequency than that of another core running a sensor task *B*, on which *A* depends in terms of data, the data acquired by task *A* might be outdated for more than one execution period. Similarly, if the core running task *A* gets a drastic decrease of the frequency during runtime task *A* will miss some of the data collected by task *B*. A strong challenge of using DVFS processing cores is when to tune the frequency and how much the tuning must be [12], [11], [29].

Due to the safety requirements, performance and energy can be sacrificed up to a certain degradation level if the safety and reliability are jeopardized. A compromise between safety , performance and energy consumption can be established to maintain safety properties while the performance and energy consumption are aimed to be optimized as much as the system safety permits. This article studies the tradeoff between the energy consumption, schedulability (as a safety property) and different performance metrics. To such an end, we propose a scheduling algorithm to arbitrarily optimize certain performance metrics, such as energy consumption and memory bottleneck, and study its impact on the rest of metrics and schedulability. Our ultimate goal is to maximize the number of performance attributes that can be optimized without necessarily degrading the rest of the attributes. The questions that this paper answers are: 1) *how multicore platforms can be used for safety critical systems without under-utilizing their resources*?; 2) *how to reduce the energy consumption and optimize the performance without affecting*

J. Boudjadar is with the Department of Engineering, Aarhus University Denmark (e-mail: jalil@eng.au.dk).

the schedulability?.

A model-based realization of our framework is made using UPPAAL, so that schedulability is formally analyzed using symbolic model checking whereas performance and energy consumption are analyzed using statistical model checking.

The rest of the paper is organized as follows: Section II introduces the systems architecture we consider. Section III describes the necessary background. Section IV is the problem statement. Section V reviews the related work. Section VI is the mathematical setting of our framework. Section VII describes our scheduling protocol. Section VIII presents the UPPAAL model-based realization. Section VIII-C is the analysis of the trade-off using a case study. Section IX concludes the paper.

II. Architecture of Multicore Scheduling Systems

Multicore embedded systems are formed by the mapping of a multi-component software system to a multicore platform. Each component in turn is formed by the parallel composition of a set of tasks, each of which represents a behavior described with real-time constraints. A multicore platform is given by a set of processing cores, local caches and shared memory. Cores can be either homogenous or heterogeneous and may feature different characteristics.

In order to enhance the processing performance of multicore platforms, modern multicore processors[1] consider a shared cache level L2 besides private caches (L1-cache). The primary reason of sharing a cache between different cores is to reduce the access requests to the main memory DRAM, and by that shorten the DRAM interference time since the interference time is strongly correlated to the number of access requests [28].

The mapping of tasks to cores can be either static or dynamic. In the dynamic mapping, a global scheduling policy is adopted where only one waiting queue is used to serve all cores. In contrast, local scheduling is used to schedule tasks statically allocated to a given core. The local scheduling is practical to implement the IMA partition-based architecture.

To reduce the memory interference of memory-intensive multicore systems, application tasks can be scheduled according to a decreasing order of the numbers of memory requests[2] they issue during a given time interval. This core scheduling policy is known by *memory-centric* [35].

The system architecture we consider is formed by a set of subsystems each of which is mapped to a given processing core. We consider a hierarchy of memories where each core has a local cache (L1) and shares a cache level (L2) and DRAM with the other cores. Each core is also characterized by a set of processing frequencies, each of which associated with an energy consumption rate. Moreover, we adopt the Asymmetric Multi-Processing (AMP) [16] scheduling architecture, where task sets (partitions) are statically allocated to specific cores. This choice is motivated by the need to prevent error propagation between the applications running on different cores. Fig. 1 depicts the overall architecture adopted throughout this paper. Our collaborative scheduling mechanism imposes the four schedulers (*S1*, *S2*, *S3*, *S4*, *CC* and *FR-FCFS* in Fig. 1) to coordinate so that tasks are efficiently scheduled to shorten memory interference. For example, when a core **C** is not allowed to access a shared memory (L2 or DRAM) due to a budget expiry [3]. The scheduler of the concerned shared memory (*CC* or *FR-FCFS*) notifies the scheduler of core **C** so that it preempts the current running task whenever it exhibits a memory access request. The scheduler of core **C** schedules another ready task not immediately requesting an access to the concerned shared memory.

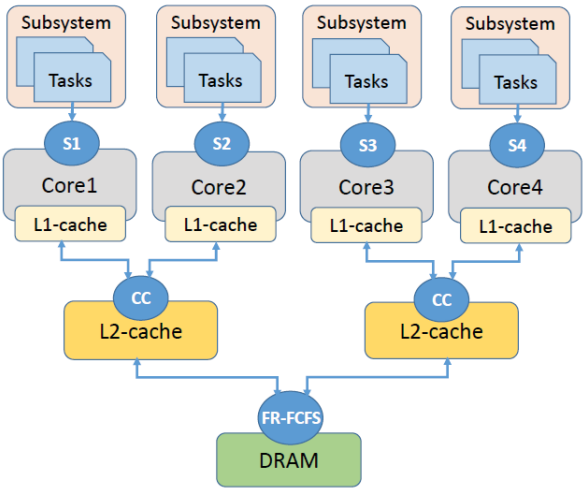


Fig. 1 Overall architecture

III. Background

This section introduces the fundamental background of our work.

A. Shared Memory Access Scheduling

The shared memory scheduling policies used in this paper are cache coloring and First Ready-First Come First Serve. Cache coloring (*CC*) policy [22], [36] controls the access to the shared cache L2 and improves the performance by mapping physical memory pages to cache pages, thus avoiding the clearance of cache pages on each context switch. The coloring algorithm sorts the concurrent access requests according to their release times. During execution, the algorithm frees the old pages as necessary in order to make space for currently scheduled applications (recoloring).

DRAM controllers in modern COTS-based systems use First Ready-First Come First Serve (*FR-FCFS*) policy [30], [21] to schedule concurrent memory accesses. FR-FCFS considers a detailed DRAM architecture structured in terms of banks, rows and columns. The access requests can target different banks,

[1]E.g. Intel Core i7, AMD FX, ARM Cortex and FreeScale QorIQ processors.

[2]Technically known by Worst Case Resource Access number (WCRA) [28].

[3]An access budget states how many accesses to a given memory a core can perform over a given time duration.

where they will be queued in the corresponding bank queue with a special preference to read requests since they cause the processor to stall. Access requests will be sorted at each bank queue first according to their readiness. Thereafter, the candidates selected from banks level will be further sorted at bus level where the earliest request gains access, i.e. the first request showing up at bus level among the requests selected by bank schedulers. If no request hits the row-buffer, older requests are prioritized over younger ones. A request hits a row buffer if it has high row buffer locality, i.e it targets a row being recently accessed.

B. DVFS and Dynamic Scheduling

Dynamic voltage and frequency scaling (DVFS) processors have been introduced as a flexible computing technology to execute systems operating on energy-limited sources and having a non static workload [29]. Such platforms have been intensively used to extend battery lifetime as energy consumption is proportional to the cores frequency, lower the hardware temperature and increase the number of applications that can use the system resources. Examples of hardware processors adopting frequency scaling are HP Proliant servers and all Intel multicore processors since Dual-core Itanium2 2004. Operating systems are accordingly extended with controllers to scale up/down the processor frequency, e.g. *cpufreq* module equipping the Linux kernel since version 3.4.

The widely used DVFS techniques scale up and down the frequency of processing cores to adjust the computational power according to the actual workload time constraints [13], [1], [29], among other factors such as thermal emergencies and power consumption constraints. In principle, during runtime whenever the workload increase reaches a level, specifying the processing capacity of the platform under a given frequency, processing cores can run faster using a higher frequency to make room for all tasks to execute before their deadlines [17]. Similarly, once a resource utilization approaches certain budget, such as energy consumption, the frequency of cores running non-CPU intensive tasks can be tuned down accordingly [25]. The tuning of cores frequency is performed on the fly using controllers and load calculators [1], [13]. However, the use of dynamic frequency scaling makes the real-time guarantees hard to be delivered. Moreover, running a core faster increases the frequency of issuing memory access requests as code instructions get executed in shorter time. This may in turn lead to longer memory bottleneck. According to our experiments, memory interference time can increase linearly with the frequency increase.

C. Statisticsal Analysis

To perform quantitative evaluation of the performance and energy consumption, we use UPPAAL Statistical Model Checker (SMC) [5]. In fact, SMC enables simulation of the system executions so that different metrics can be statistically evaluated. In terms of specification, UPPAAL enables describing systems as a set of concurrent processes using a state-transition formalism called *timed automata*. UPPAAL SMC enables to analyze execution traces, that are randomly selected, with respect to different quantitative properties. We can summarize the main features of UPPAAL SMC as follows:

- Stopwatches [10] are clocks that can be stopped and resumed without a reset. They are very practical to measure the execution time of preemptive tasks.
- Probability evaluation `Pr[bound] (P)` for a property `P` to be satisfied within a given simulation time and/or number of runs specified by `bound`.
- Simulation and estimation of the expected minimum or maximum value of expressions over a set of runs, `E[bound](min:expr)` and `E[bound](max:expr)`, for a given simulation time and/or number of runs specified by `bound`.

Statistical model checking does not provide complete certainty that a property is satisfied, but only verifies it up to a specific confidence level [14], given as an analysis parameter. Beside to statistical features, UPPAAL enables the verification of safety and liveness properties using symbolic model checking technique. Properties can be expressed using Computation Tree Logic (CTL).

IV. Problem Statement and Contribution

A challenge of the multicore architecture depicted in Fig. 2 is how to design feasible schedules that lead to less energy consumption, short memory interference time and high utilization of processing cores. These different metrics are correlated in the way that improving a performance metric degrades other metrics. An example is the cores utilization for which an enhancement via frequency tuning leads to higher frequency of issuing memory access requests [34], [9], which in turn increases the accumulated waiting time to access DRAM and worsens the memory bottleneck.

Let us assume the runtime scenario depicted in the snapshots of Fig. 2 Initially, core $C1$ is performing access to DRAM (snapshot (a)). Meanwhile, core $C2$ and $C3$ successively perform access requests and join the DRAM queue (snapshots (b) and (c) are omitted). In snapshot (d), core $C4$ performs a DRAM access request while the other core requests are pending. While core $C4$ is waiting for the access to DRAM, which is going to be relatively long as $C4$ is the last element of the access, the following facts hold:

- The waiting time is accumulated to the response time of the task currently running on $C4$.
- During the waiting time, the core is running but not effectively used.
- During waiting time, the core is consuming energy while doing effective work.

This paper proposes a dynamic scheduling control to optimize these three points and studies their mutual impact. To such an end, we preempt the task currently scheduled on any core once it performs a memory request that is not immediately granted (task $T1$ in Fig. 2 (a)), because either the request is preceded by many others pending requests or the corresponding core has consumed its entire access budget for the current time interval. Such a task will be moved to a new scheduling queue (`AccessDueQueue`) pending for

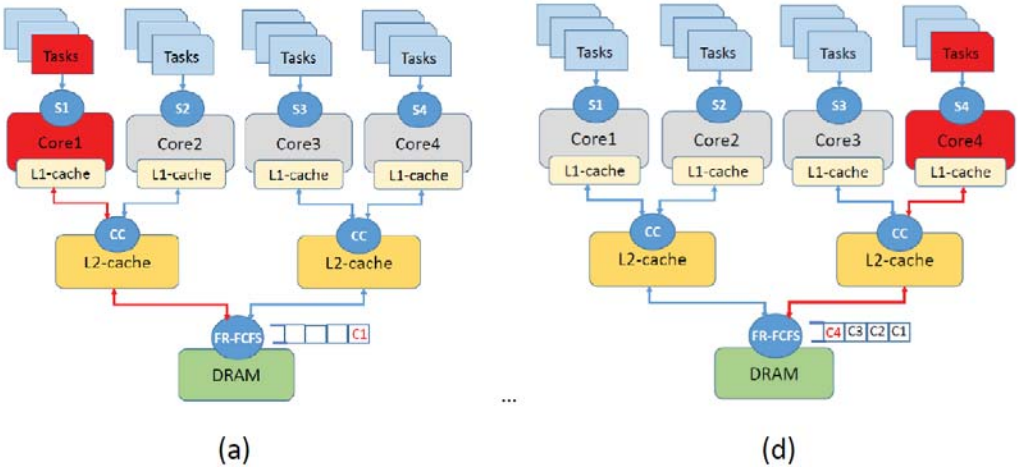


Fig. 2 Runtime example

an access request. Another ready task from the regular queue of such a core (task $T2$ in Fig. 2 (b)), can immediately be scheduled as the core just becomes available, i.e. not stalling. In a similar way, if the newly scheduled task exhibits an access request to a shared memory while the target memory queue is very dense, such a task will immediately be preempted and moved to `AccessDueQueue` thus leading another ready task to run. Whenever the access request of a preempted task is granted, the corresponding task will be scheduled immediately (task $T1$ in Fig. 3 (c)) causing the current running task (task $T2$ in Fig. 3 (c)) to be preempted. This is to respect the core's scheduling policy as the running task ($T2$) is not supposed to run before the task resuming right now ($T1$). In a similar way, when the access request of another preempted task (located in `AccessDueQueue`) is granted while the current running task just resumed from the interference queue, such a task will be inserted in the core's regular queue right after the last task moved from `AccessDueQueue` to regular queue. Accordingly, we enhance the effective utilization of cores and reduce the energy consumption of processing cores. We use DVFS to improve the performance further where cores frequency is tuned following the *interference*, i.e. the state of DRAM and L2 access queues so that we do not worsen memory bottleneck. We implement this scheduling strategy as an algorithm to tune the control parameters and shuffle t he p erformance o bjectives. W e s tudy t he i mpact of such scheduling strategy on the different metrics (performance, energy) of the system.

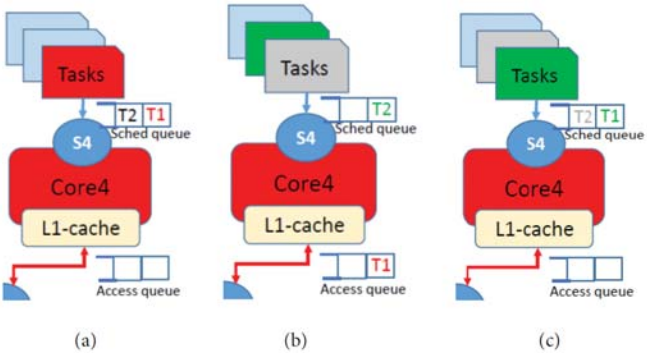


Fig. 3 Scheduling example using the proposed strategy

V. Related Work

Energy-aware and performance-aware scheduling protocols for multicore systems have been studied thoroughly in the literature [25], [13], [24], [2], [27], [18], [29], [9] with the goal reduce energy consumption and improve performance of such systems while delivering high schedulability guarantees. In a similar way, different algorithms have been proposed to make the use of cores frequency scaling more profitable with respect to energy consumption and performance [12], [6]. However, this makes real-time guarantees hard to be delivered as tuning is applied on per-core basis which may affect runtime predictability and synchronization between application tasks [25], [8]. Moreover, the energy consumption and different performance metrics are correlated in the way that improving one metric may degrade others [34].

Li *et al.* [24] propose an energy-aware algorithm to schedule dependent tasks. The algorithm consists in calculating an optimal task-to-core assignment and estimating the proper voltage to execute a given task. Similarly, Lin *et al.* [25] proposes 2 task scheduling algorithms to leverage per-core DVFS and achieve a balance between performance and energy consumption. In both [24] and [25], the proposed algorithms minimize the number of voltage/frequency transitions and rather migrate tasks between cores. Although these techniques are promising, the migration operation contributes the memory bottleneck and degrades the performance as more accesses to DRAM are need for the context switch.

Datta *et al.* [13] introduced 2 priority-based scheduling algorithms to reduce energy consumption and increase performance. Priorities are calculated based on either cache miss ratio or context switch of processes. However, using performance and energy as main criteria for the tasks scheduling might lead to deadlines miss.

Subramanian *et al.*. [33] introduced a performance-driven scheduling algorithm, called (MISE-Fair), to minimize the

maximum slowdown delays experienced by tasks, i.e. the delays due to waiting for access to shared resources. In essence, MISE-Fair estimates the slowdown of each application and redistributes the memory bandwidth to reduce the slowdown of the most slowed-down application. However, minimizing the DRAM-related slowdown of tasks may impact scheduling at the cores level.

The authors of [35] propose a *global* memory-centric scheduling algorithm and study the execution slowdown while varying the number of cores. A conclusion is that the memory-centric policy can schedule twice as many tasks compared to processor-centric scheduling, however this may lead to drastic energy consumption of the processing cores.

In this paper, we introduce a scheduling algorithm to tune different performance parameters on the fly, each at a time, and study the impact on the rest of the performance metrics and energy consumption. We mechanize our system model and the scheduling algorithm in Uppaal to provide a ground for the trade-off analysis where statistical model checking is used to examine energy consumption and performance whereas symbolic model checking is used to verify schedulability.

VI. Systems Specification and Analysis

This section presents a formal specification of our system model and how to calculate the different metrics: energy consumption, performance and response time.

A. System Settings

We consider an application model $\mathcal{A}$ formed by a set of task sets $\{\mathcal{T}_1, \mathcal{T}_2, ...\}$, each of which is mapped to a given core of a multicore platform $\mathcal{P}$.

A task set, or shortly a component, $\mathcal{T} = \{T_1, .., T_n\}$ is a set of periodic tasks. A task models the execution of a real-time process for both processing and memory access. We consider 2 attributes α_c and α_m where:

- α_c corresponds to the maximum number of successful access requests (hits) to L2;
- α_m is the number of DRAM access requests (corresponds to L2 miss) performed by a given task.

Given that read and write access requests have different impact on processor time (stalling), we denote each of these attributes with r for read and w for write, i.e. α_c^r, α_c^w, α_m^r and α_m^w to distinguish between read and write requests. Such attributes are identifiable using program/cache analysis tools [15] on a given platform architecture. We formally define access requests to shared memories as follows:

Definition 1 (Memory access requests): The access request of a process to a shared memory is given by $req = \langle \rho, RW, \psi \rangle$ where:

- $\rho \in \{L2, DRAM\}$ is a pattern stating to which memory the access hits.
- RW states whether it is a read (r) or write (w) request.
- Our task model triggers access requests non-deterministically during runtime, we use thus ψ to store the issue time of each request once is effectively issued. ψ is initially empty.

For the behavior of each task, we use α_c^r and α_c^w , respectively α_m^r and α_m^w, as numbers of read and write accesses to L2, respectively DRAM. Implicitly, each access request to L2 is preceded by an access to the core local cache (L1 miss).

Now we specify the task attributes needed to perform an efficient real-time scheduling.

Definition 2 (Task structure): A periodic task T is given by $\langle p, o, \tau, \alpha_c^r, \alpha_c^w, \alpha_m^r, \alpha_m^w, d, \epsilon \rangle$ where:

- p is the task period, and o is the periodic offset.
- τ is the pure worst case execution time (WCET) of T when executing on a processing core operating the lowest frequency, i.e. τ does not include the time to fetch data from shared memories.
- $\alpha_c^r, \alpha_c^w, \alpha_m^r$ and α_m^w are the number of access requests described earlier.
- d is the relative deadline and ϵ is the priority level associated to T.

As part of the AMP architecture, the tasks of a given component are allocated to the same processing core. The platform model $\mathcal{P}$ consists of a set of processing cores $\mathcal{C} = \{C_1, C_2, ...\}$, one shared cache level (L2) and a shared DRAM. A processing core is given as follows:

Definition 3 (Processing cores): A processing core C is given by $\langle \langle f_1, .., f_n \rangle, \langle v_1, .., v_n \rangle, \beta_m, \beta_c, \Phi, H \rangle$ where:

- $\langle f_1, .., f_n \rangle$ is a set of processing frequencies.
- $\langle v_1, .., v_n \rangle$ is the voltage rates corresponding to the individual frequencies.
- β_m and β_c are budgets stating the maximum access requests of a core, to L2 and DRAM respectively, per second.
- Φ is the scheduling policy adopted by C.
- H is the local cache abstracted as a static access time.

The scheduling function $\Phi_i : \mathcal{T}_i \times \mathcal{T}_i \times \mathbb{R}_{\geq 0} \rightarrow \mathcal{T}_i \cup \{\texttt{none}\}$ of a core C_i identifies, among the ready tasks $\mathcal{T}_i$ allocated to C_i, which one has priority to be scheduled at any point in time. The scheduling of a task may imply a preemption of of another task. In case none of the tasks is ready, Φ returns `none`. Φ can be implemented for both static and dynamic priority scheduling algorithms.

Definition 4 (Shared memories): A shared memory, L2 and DRAM, is given by $\langle \Phi_x, \delta_x \rangle$ with $x \in \{m, c\}$, where Φ_x is the access scheduling function and δ_x is the effective access time, i.e. the time duration of fetching data from a physical address once the access is granted.

The memory scheduling function $\Phi_x : \mathcal{C} \times \mathcal{C} \times \mathbb{R}_{\geq 0} \rightarrow \mathcal{C} \cup \{\texttt{none}\}$ states which of the cores can be granted access to a shared memory, L2 or DRAM, at any point in time. In case none of the cores is currently performing an access request, Φ returns `none`.

B. Calculation of Energy Consumption

We analyze the energy consumption of the multicore platform using statistical model checking. To this end, we consider a set of independent runs $Runs = \{\pi_1, \pi_2, \ldots\}$,

randomly generated according to a probabilistic semantics [14]. A run π of a system S is an infinite sequence:

$$\pi = s_0(t_0, e_0)s_1(t_1, e_1)\dots s_n(t_n, e_n)\dots$$

where each state s_i gives information about a system configuration including the state of each task (e.g. idle, ready, running, blocked) and resource (e.g. idle, occupied, blocked, *etc*); s_0 is the initial state. Each e_i indicates an event (triggering, queued, tuning-up, tuning-down, released, completing, *etc*) signifying a transition from state s_i to s_{i+1} where at least one characteristics of either a core, a task or a shared memory changes. Timestamp t_0 indicates the time from system initiation until event e_0. Every subsequent timestamp t_i (with $i \geq 1$) indicates the duration between events e_{i-1} and e_i.

Definition 5 (Energy consumption): The energy consumed by a given core C_j for a run π is the accumulation of energy consumed at each state of the run, which is in turn a function of the core frequency operated at the state [6]. Since each state runs *only one* frequency of C_j, as a tuning up/down leads to another state in the trace π, the energy consumption of C_j is formally given by:

$$E_\pi(C_j) = \sum_{k=0}^{\infty} 1/2\ \chi\ (volt(s_k))^2\ freq(s_k)\ t_k$$

where χ is the core capacitance; functions $volt(s_k)$ and $freq(s_k)$ return the core voltage and frequency at a given state s_k. t_k is the time duration, from the trace definition, upon which a core maintains its frequency and voltage.

For a given time duration t, the energy consumption of a core C_j is given by $\sum_{k=0}^{x} 1/2\ \chi\ (volt(s_k))^2\ freq(s_k)\ t_k$ where $\sum_{k=0}^{x} t_k = t$. Accordingly, the energy consumed by a set of cores is the sum of individual cores consumption. To obtain reliable results, one needs to consider a large set of runs.

C. Calculation of the Response Time

To calculate the response time of a given task T_j on a given run π we introduce the following:

- $Released_\pi^{T_j}(s)$ indicates whether T_j is *released* for a new period at state s or not.
- $Done_\pi^{T_j}(s)$ indicates whether the execution of T_j is *done* at state s or not.

Since a task can maintain a status through different states, e.g. a task can be ready over a set of states while waiting to be scheduled, one needs to identify what is the first state at which the given status is obtained.

Definition 6 (Response time): The response time $R_\pi(T_j)$ of a task T_j is the maximum duration between the release and termination of each execution period. Formally,

$$R_\pi(T_j) = \max(\sum_{k=x}^{y} t_k \mid \ Released_\pi^{T_j}(s_x)\ \wedge\ Done_\pi^{T_j}(s_y) \wedge \forall s \in\ ]s_x, s_y[\ (Released_\pi^{T_j}(s) \vee\ Done_\pi^{T_j}(s)) = false)$$

We define the final response time $R(T_j)$ of task T_j to be the maximum of the response times obtained on individual runs, i.e. $R(T_j) = \max_i R_{\pi_i}(T_j)$.

Given that schedulability is a safety property, we use symbolic model checking to analyze it. In fact, we explore the whole state space (all potential runs) and check at each state whether there is a task missing its deadline or not.

D. Performance Calculation

In this paper, we use cores utilization and memory interference as performance metrics [9]. To quantify such metrics for a given run π, we define the following predicates:

- $InUse_\pi^s(C) \in \{0, 1\}$ is a predicate that indicates whether core C is being used in state s of run π. For the sake of implementation, we consider Boolean values as integers either 0 or 1. In fact, a processing core is in use if it is running a workload, either effective executing or stalling due to a shared memory access.
- $Issued_\pi^C(req)$ indicates when the access request *req* is effectively triggered by core C, i.e. at which state of π.
- $Granted_\pi^C(req)$ indicates at which state of π the access request *req*, performed by C, is performed.

Definition 7 (Processing core utilization): The utilization of a processing core C on a given run π is the accumulated time of core C being active. Formally, the core utilization $U_\pi^{\mathcal{L}}(C)$ up to a time bound $\mathcal{L}$ (simulation length) is given by the following:

$$U_\pi^{\mathcal{L}}(C) = (\limsup_{t \to \mathcal{L}} \frac{\sum_{t_{i+1} \leq \mathcal{L}} ((t_{i+1} - t_i) \times InUse_\pi^{s_{i+1}}(C))}{\mathcal{L}}) \times 100$$

The average utilization of the processing core C for a set of runs *Runs* will be the accumulated individual utilizations on the number of runs:

$$U_{Runs}^{\mathcal{L}}(C) = \frac{\sum_{j=1}^{j=|Runs|} U_{\pi_j}^{\mathcal{L}}(C)}{\mid Runs \mid}$$

Definition 8 (Interference of memory access requests): The interference of an access request to a shared memory is the accumulated delay for the request since its release to the time point it is granted. Formally, for a given access request *req* performed by core C in a run π, we define the interference of *req* as follows: $ReqDelay_\pi^C(req) = Granted_\pi^C(req) - Issued_\pi^C(req)$.

Similarly, the maximum delay of the requests Req_C issued by a core C on a run π is given by:

$$ReqDelay_\pi^C = \max(ReqDelay_\pi^C(req \mid req \in Req_C))$$

Accordingly, the average of maximum interference delays of core C for the set of runs *Runs* is defined as follows:

$$ReqDelay_{Runs}^C = \frac{\sum_{j=1}^{j=|Runs|} ReqDelay_{\pi_j}^C}{\mid Runs \mid}$$

Memory interference is critical to the calculation of the tasks' response time and schedulability analysis given that memory interference time contributes to the tasks

execution time. We distinguish between $L2\_ReqDelay^{C}_{Runs}$ and $DRAM\_ReqDelay^{C}_{Runs}$, as the interference delays of a given core C to access L2 and DRAM respectively, given that access requests to L2 and DRAM do not have the same interference delays, i.e. DRAM access is considerably longer compared to the access to shared cache L2.

VII. Scheduling and Cores Frequency Tuning

A task is scheduled according to the scheduling policy of the core to which is assigned, once it runs an access request the corresponding core performs a request and waits for the request to be granted. In contrast to read requests, a core does not stall with a write access request and gets immediately unlocked. Once the max budget (number of access requests allowed for a time duration) of the core is reached, the core is not able to perform access requests. Rather the running task will be preempted due to the access request, and will be placed in another scheduling queue `AccessDueQueue`. This leads to exploit the core time in executing other ready tasks rather than waiting for a long interference time. The core obtains then another ready task to run, however once the current task needs to perform a request while the core budget is still not replenished that task will as well be preempted due to access request and placed in the same queue `AccessDueQueue`. Once the core budget is unlocked, for a new period, the core scheduler preempts the current running task and schedules the tasks located in `AccessDueQueue`, using a FIFO policy. The core keeps running all tasks in `AccessDueQueue` before switching to the regular scheduler queue where the adopted scheduling policy will be used then. To sum up, while the core access budget is not reached yet tasks are purely scheduled according to the core scheduling policy. Once an access budget is reached, the scheduling will be driven by both core policy (to select from ready tasks waiting for the core), and whether or not a task needs to perform an access request (as a preemption criterion). This in fact leads to shorter interference delays as we decrease the frequency of issuing access requests over time.

Fig. 4 depicts an example of how our collaborative scheduling mechanism works. Initially, all ready tasks join the ready queue whereas the task having priority (in this case **T1**) runs first. Given that the core budget is expired (scenario step (b)), whenever **T1** exhibits an access request it gets immediately preempted and moved to `AccessDueQueue`. Similarly, **T2** is scheduled but at the first attempt to access shared memory the core scheduler preempts it. Task **T3** is scheduled, meanwhile the core budget is replenished (scenario step (c)). Thus, **T3** is immediately preempted and **T1** is scheduled to complete its execution (scenario step (d)). The core then keeps executing the tasks stored in `AccessDueQueue` before scheduling any task from the ready queue.

While alternating the scheduling mechanism between core-centric and memory/L2-centric for a given core, the core frequency will be scaled down to reduce energy consumption when stalling for an access request, having a light workload or having a budget expired. The frequency is tuned up to absorb

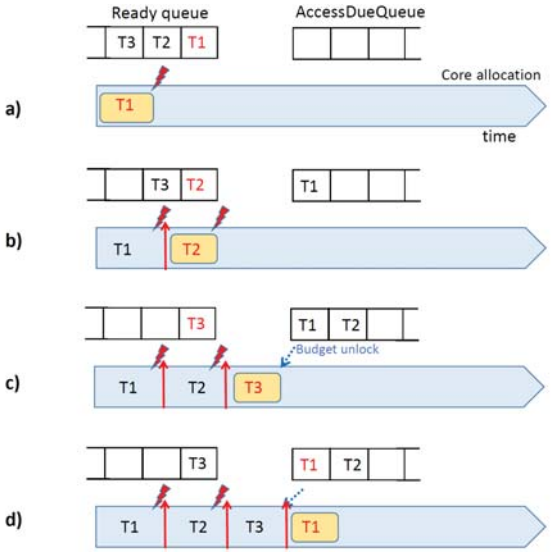


Fig. 4 Collaborative scheduling

the backlog created by moving tasks to *AccessDueQueue*, or in case there is a potential for a ready task to miss its deadline. Given a core C running a frequency f_i at a time instant t, the tuning of C is performed as follows:

$$Tuning(C,t) = \begin{cases} f_1 & \textbf{if } Inactive(C,t) \textbf{ or } Stall(C,t) \\ f_{i+1} & \textbf{if } f_i \neq f_n \textbf{ and } heavy(workload,t) \\ f_{i-1} & \textbf{if } (f_i \neq f_1 \textbf{ and } not(heavy(load,t)) \\ & \quad \textbf{and } budgetExpired(C,t)) \end{cases}$$

The core workload (`workload`) includes both the load (`load`) formed by regular ready queue and the tasks preempted due to budget expiry (located in `AccessDueQueue`). We consider tuning up/down with one level only, however the tuning controller updates the core frequency more than once per time, iteratively, if the conditions still hold [4]. The auxiliary functions used above are given by:

$$Inactive(C,t) = \begin{cases} true & if\ \forall T_i\ T_j\ \Phi(T_i,T_j,t) = \texttt{none} \\ false & otherwise \end{cases}$$

$$Stall(C,t) = \begin{cases} true & if\ \exists C'\ \Phi_x(C,C',t) = C' \\ false & otherwise \end{cases}$$

$$load(C,t) = \sum_{T_i \in \mathcal{T}} (Left_{WCET}(T_i,t)/freq(t))$$

The load of a given core is formed by the remaining WCET of ready tasks using the current core frequency $freq(t)$. Since WCET is given in terms of the lowest frequency level, the effective execution time will be a division of WCET on the current frequency (normalized to the lowest frequency level).

$$workload(C,t) = load(C,t) + \sum_{T_i \in \mathcal{T}} Left_{WCRA}(T_i,t) * a^r()$$

The workload is formed by a load together with how many access requests left for each task multiplied by the current worst case response times of the shared memories (L2 or

DRAM) estimated online using the memory response time function $a^r()$ [7].

$$heavy(X,t) = \begin{cases} true & if \frac{X}{\sum_{T_i \in X} (d_i - t\%p_i)} \geq 1 \\ false & otherwise \end{cases}$$

$\mathcal{T}$ is implicitly the task set assigned to the core C under analysis. Basically, function $heavy(X,t)$ compares the load/workload X to the time left to deadlines d_i, from the release of the current periods ($t\%p_i$). One can see that we overload function $freq()$, so that the parameter can be a time instant t or a state s in a runtime trace π.

VIII. Uppaal Modeling and Trade-off Analysis

Our system model aligns with [35], but we adopt *local* scheduling (AMP) and spread out memory access requests non-deterministically during task executions rather than using dedicated phases. This is in fact much realistic as actual systems do not have exact time points to fetch data.

A. Uppaal Modeling

We have built a model-based framework, using Uppaal, to implement the theory introduced in Section VI. Such a model-based setting enables schedulability analysis and implements our scheduling technique, energy consumption and frequency tuning calculations. The system application is given by a set of periodic task sets, each of which is statically assigned to a given core. Cores share both L2 cache and DRAM. We model tasks with explicit read and write access numbers for shared cache L2 and DRAM. We distinguish between read and write access requests to shared memories as read actions make cores stall, while write actions are not blocking and can be performed using dedicated buffers. The system is formed by a parallel composition of the processes representing the different entities: tasks, processing cores, scheduler, L2 and shared memory. For reconfigurability purposes, the processes are made parametrized so that an instance of a given template can easily be created by just providing actual parameters.

We only demonstrate the task model as it is the main process of our model behavior.

The task model is depicted in Fig. **??**. The task starts at location Init and moves to to location Ready, upon the expiry of a potential offset, to request the core (*C*) it is mapped to. In order to make our models flexible and reusable, the core identifier is a parameter of the task mapping. The task waits to be scheduled at location WaitSched unless the deadline is reached by which it moves to location DeadlineMiss. Once scheduled at location Run, a task starts its execution and non-deterministically triggers access requests to L2 and DRAM. One can see that at location Run, the progress rate of the clock measuring the task execution time WCET is given in terms of the current core frequency (*execTime[tId]'==core[c].CurFreq*), thus the faster the core is the shorter the execution will be. A task can be preempted either when it performs an access request to a shared memory while the core access budget is expired (location APreempted), or another higher priority task preempts it during regular execution (WaitSched). Either cases, the task moves back to location WaitSched. However, it can only be located in one waiting queue, either *queue* or *Aqueue*. If the task execution, for both WCET and memory accesses, terminates before deadline the task then moves to location Done. Otherwise, a deadline miss is reported.

B. Schedulability and Energy Consumption Analysis

According to the task model described earlier, whenever a process misses its deadline it joins immediately the location DeadlineMiss. Thus, the schedulability analysis is performed using symbolic model checking and simply checks whether any task can reach its own DeadlineMiss location. To quantify on all tasks regardless of their identifiers we use the following CTL query supported by Uppaal:

$$\forall[]\ !error \quad (1)$$

To analyze the energy consumption of a given core, we use statistical model checking (SMC) where we run a system execution (simulation) several times (X), each of which lasts for Y time units, and accumulate the energy consumption measured using the clock variable *Energy[cId]*. Globally, the larger X and Y are the more accurate the results will be because more execution scenarios will be explored. The energy consumption of an individual core C can be displayed in terms of a probability distribution using the following SMC query:

$$E[clk <= Y; X](max : Energy[C]) \quad (2)$$

Similarly, the number of frequency changes performed by a given core during certain simulation duration of Y time units can be tracked using the variable *FreqChanges[C]*, and displayed as a probability distribution using the following SMC query:

$$E[clk <= Y; X](max : FreqChanges[C]) \quad (3)$$

The response time is statistically calculated in similar way, then can be compared to the task deadlines as a cheap schedulability analysis. If each response time does not exceed the corresponding deadline then the system is *likely* schedulable; thus a formal and expensive analysis using symbolic model checking can be performed to make absolute certainty.

For performance and energy consumption optimization, one can use multi-objective Pareto frontier to compare the energy consumption, memory interference and number of cores tuning for different configurations and identify the system configuration achieving the best analysis outcome. An optimal configuration can be achieved if one establishes a priority between the aforementioned criteria. Through out this paper, schedulability is implicitly considered as a primary criteria for the deployment of an application model on a given platform.

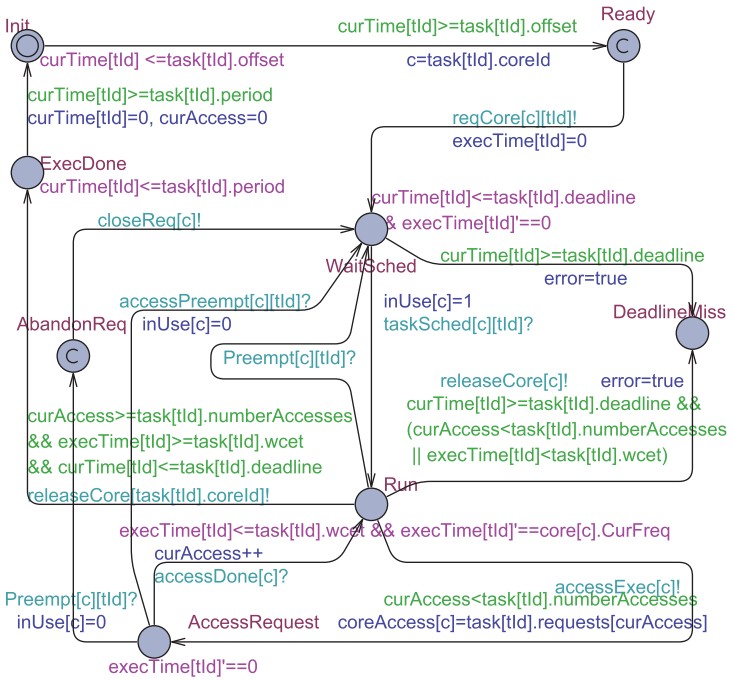


Fig. 5 Task template model

TABLE I
ATTRIBUTES OF THE AVC TASKS

Task	Prd	Offset	WCET	$WCRA_c$	$WCRA_m$	Dln
T_1	520	30	80	(40; 67)	(11; 8)	520
T_2	800	20	60	(32; 87)	(18; 10)	800
T_3	1000	200	30	(29; 78)	(12; 27)	1000
T_4	4000	0	60	(123; 64)	(19; 12)	4000

C. Trade-off Analysis: Case Study

To study the trade-off between energy consumption, performance and schedulability we analyze an avionic system component running on two processing cores each of which can operate four different frequency levels. The numbers of L2 and memory access requests are synthetically estimated according to [36]. The system description is shown in Table I, where $WCRA_c$ and $WCRA_m$ are given in terms of (reads; writes), and all time values are in milliseconds. Schedulability analysis is formally analyzed using symbolic model checking, whereas energy consumption and performance analysis are performed using SMC.

The SMC experiments are performed on 10^3 runs, each of which lasts for 10^4 time units. Fig. 6 shows the probability distribution of the energy amount consumed by core1 during the simulation time. One can see that values in [2331,2342] are the most likely candidates as they have the highest probability (0.94) whereas the worst case energy consumption value is 2444. Similarly, the number of frequency changes performed by core1 is depicted in Fig. 6 in terms of a probability distribution. One can see that the number of frequency changes varies from 22 to 102, while the most likely value is 34.

Now, we run four different simulation experiments where on each simulation the processing cores run constantly a different frequency $f_i \in \{f_1, .., f_4\}$. We analyze accordingly the

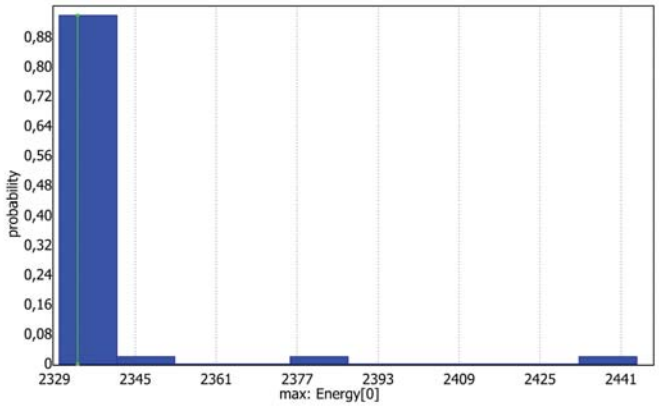


Fig. 6 Probability distribution of the energy consumed by core1

underlying schedulability, effective utilization of processing cores, memory interference and energy consumption. Fig. 7 depicts how the frequency of processing cores impacts the system schedulability. One can see that when cores run continuously the frequency level f_1 or f_4, the system is not schedulable (below 1.0). This is in fact due to tasks either missing deadlines due to slow execution (case of f_1) or having a large memory interference time (case of f_4).

Fig 8 depicts the impact of frequency scaling on the average energy consumption of processing cores. It is trivial that the energy consumption increases linearly with the increase of processing frequency. However the linear ratio gets smaller and smaller when the frequency reaches level F_3.

Fig. 9 demonstrates the relationship between the processing cores frequency and the memory bottleneck. In fact, the memory interference increases exponentially with the

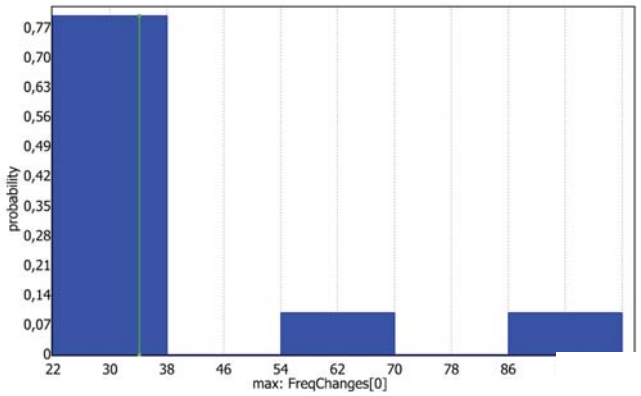


Fig. 7 Probability distribution of frequency changes of co

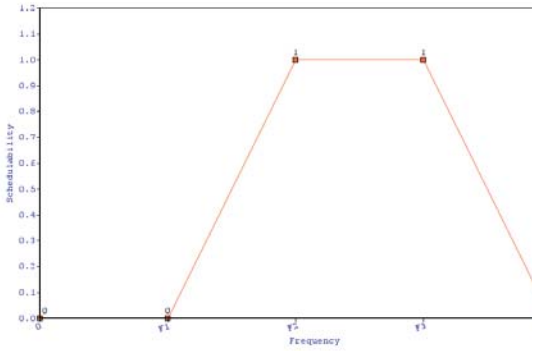
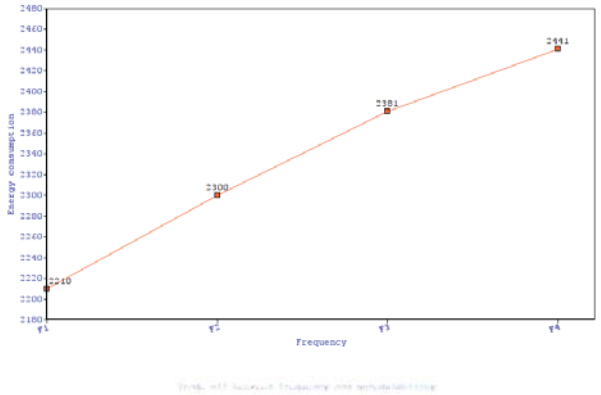


Fig. 8 Impact of cores frequency on energy consumptio



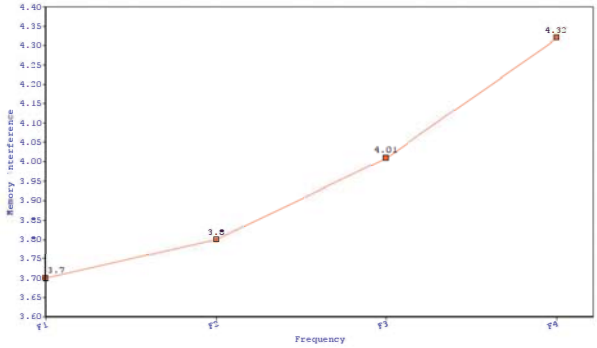


Fig. 10 Impact of cores frequency on memory interference

increase of processing frequency. This is due to the increase of processing frequency leads to higher frequency of issuing memory requests and thus longer accumulated interference time to serve such requests.

The relationship between processing cores frequency and the effective utilization of cores is depicted in Fig. 10. By effective utilization we mean the core utilization without stalling time. The overall effective utilization percentage decreases linearly, with a slower rate, with the increase of processing cores frequency. This is in fact due to higher processing frequencies lead to longer memory interference time and thus larger stalling time for cores.

D. Comparison and Discussion

In this section, we run the earlier case study with and without the collaborative scheduling algorithm (CSA) [23] and compare the analysis outcomes. When running the conventional scheduling protocol EDF, the scheduler we consider does not preempt a task when it performs an access request to a shared memory, rather it let the processing core stalls until the request is granted. In contrast, CSA protocol preempts a task when it performs a request to a shared memory having a crowded queue, and rather a computation task is scheduled so that the processing core is used to execute an available task than idling for a memory access. The metrics we compare are energy consumption, memory interference, utilization of individual cores and tasks response time.

Table II shows some of the comparison results obtained for a simulation time of 10^4 units. Having a longer interference time for processing cores with CSA, compared to regular scheduling without CSA, does not mean that the cores are stalling longer but only the requests issued by the cores take longer to get granted. While a request is waiting to be granted, the issuing core switches to another task to perform other computations. By preempting a task when it performs a memory access request and allocating the underlying processing core to another ready task, we reduced the stalling time and thus improving the effective exploitation of processing cores. This would absolutely lead to create room for new tasks to be deployed on the processing cores. As shown in Table II, executing a workload using our CSA technique reduces the utilization of $Core_1$ with (28.10% to 38.03%) and $Core_2$ with (4.24% to 8.65%), compared to classic scheduling settings. In fact, the higher the number of requests a task set can issue the larger the core utilization gain will be. On the other hand, the higher the number of requests a task set can issue the worse the memory interference will be. This is because the memory interference is correlated to

TABLE II
PERFORMANCE COMPARISON

Metrics	With CSA		Without CSA	
	$Core_1$	$Core_2$	$Core_1$	$Core_2$
L2 interference	2.74	2.49	2.32	2.42
DRAM interference	1.98	1.96	1.86	1.89
Utilization	28.10	4.24%	38.03%	8.65%
Energy consumption	2332	1864	6107	2648

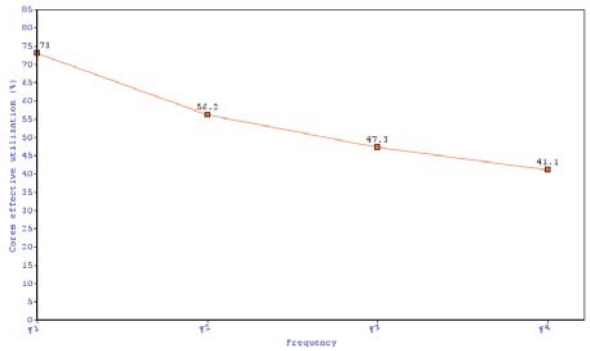


Fig. 11 Impact of cores frequency on their effective utilization

the frequency of issuing access requests.

Aligning with the cores utilization, the energy consumption of processing cores has been drastically reduced thanks to our collaborative scheduling algorithm. Table II shows that $Core_1$'s energy consumption is reduced by 61.8% (from 6107 to 2332) while the energy consumption of $Core_2$ is reduced by 29.7% (from 2648 to 1864). Fig. **??** depicts a comparison of the accumulated energy consumption of $Core_1$, to run the same workload, with and without CSA scheduling. Initially and until time point 107, both configurations (with and without-CSA) consume the same energy amount. As soon as CSA algorithm starts preempting $Core_1$ to avoid stalling, the plots start diverging. At time instant 280, all tasks execution and access requests are satisfied for the current periods. Accordingly, $Core_1$ is running a very low frequency by which the energy consumption is barely increased until time instant 300. The energy consumption plots keep diverging even though they show the same progress patterns.

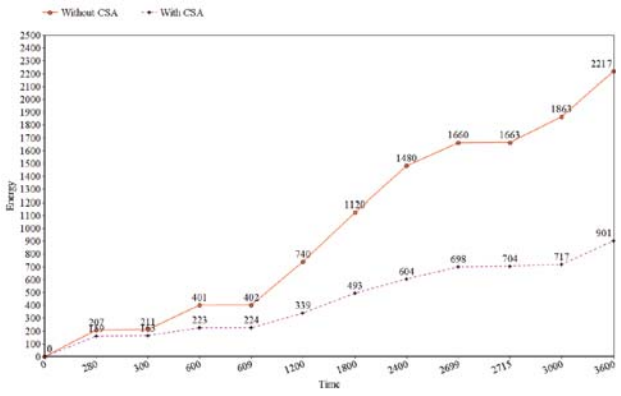


Fig. 12 Energy consumption comparison for $Core_1$

The resulting response time of tasks when running CSA is slightly longer than the response time when running regular scheduling. The response time of task T_1 when using CSA is 11.2% longer than the regular response time. On the other hand, the response time of task T_2 is reduced with 1.9%. According to our experiments, the lower the task priority is the smaller the response time difference between with-CSA and without-CSA will be. This is in fact because a lower priority task is mostly located towards the tail of the core's ready queue. Thus, when such a task runs an access request and blocks there is no other ready task to which the processing core can be allocated to do actual computation. Hence, the task response time does not differ largely when using CSA compared to classic scheduling settings.

The analysis outcomes of the case study show a trade-off between the energy consumption and different performance attributes. Reducing the cores utilization using CSA leads to reduce the energy consumption but it increases the frequency of issuing access requests which in turn crowds the shared memory queue. Thus, resulting in longer memory interference which might affect the tasks response time, in particular for memory-intensive applications, by which a deadline can be missed. One has to find a threshold to determine the right access-due-preemption time point, for example if there is less than certain number of access requests waiting in the DRAM queue the processing core issuing an access request will be better stalling and consuming a low voltage rather than swapping to another ready task, by which reducing the effective concurrent access requests to shared memories.

IX. Conclusion

This paper presents a theory and a model-based implementation to study the trade-off between schedulability, performance and energy consumption of multicore systems. The processing cores we consider share different levels of cache and memory, and run dynamic voltage and frequency scaling. The performance metrics considered are memory interference, response time of tasks and effective cores utilization. Our models have been mechanized in Uppaal where schedulability is analyzed using symbolic model checking, while energy consumption and performance are analyzed using statistical model checking.

To demonstrate the trade-off between the different metrics we consider, we have analyzed an avionic system component as a case study. Our analysis results show that the processing speed (cores frequency) is a decisive element for the rest of the metrics as it either leads to deadline miss, if too slow, or to a large memory bottleneck and stalling time of cores, if too high.

As a future work, we plan to implement a decision making process in order to assist the scheduling protocol finding the optimal configuration of the difference performance metrics and energy consumption.

References

[1] G. M. Almeida, R. Busseuil, E. A. Carara, N. Hbert, S. Varyani, G. Sassatelli, P. Benoit, L. Torres, and F. G. Moraes. Predictive dynamic frequency scaling for multi-processor systems-on-chip. In *ISCAS'11*, pages 1500–1503, 2011.

[2] M. Bambagini, M. Marinoni, H. Aydin, and G. Buttazzo. Energy-aware scheduling for real-time systems: A survey. *ACM Trans. Embed. Comput. Syst.*, 15(1):7:1–7:34, Jan. 2016.

[3] S. K. Baruah, L. Cucu-Grosjean, R. I. Davis, and C. Maiza. Mixed Criticality on Multicore/Manycore Platforms (Dagstuhl Seminar 15121). *Dagstuhl Reports*, 5(3):84–142, 2015.

[4] J.-P. Bodeveix, A. Boudjadar, and M. Filali. An alternative definition for timed automata composition. In *Automated Technology for Verification and Analysis*, pages 105–119, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[5] A. Boudjadar, A. David, J. H. Kim, K. G. Larsen, M. Mikucionis, U. Nyman, and A. Skou. Statistical and exact schedulability analysis of hierarchical scheduling systems. *Sci. Comput. Program.*, 127:103–130, 2016.

[6] A. Boudjadar, A. David, J. H. Kim, K. G. Larsen, M. Mikuionis, U. Nyman, and A. Skou. A reconfigurable framework for compositional schedulability and power analysis of hierarchical scheduling systems with frequency scaling. *Science of Computer Programming*, 113:236 – 260, 2015.

[7] A. J. Boudjadar, A. David, J. H. Kim, K. G. Larsen, M. Mikucionis, U. Nyman, and A. Skou. Degree of schedulability of mixed-criticality real-time systems with probabilistic sporadic tasks. In *2014 Theoretical Aspects of Software Engineering Conference - TASE*, pages 126–130, 2014.

[8] J. Boudjadar, A. David, J. Kim, K. Larsen, U. Nyman, and A. Skou. Schedulability and energy efficiency for multi-core hierarchical scheduling systems. In *Proceedings of the European Congress on Embedded Real Time Systems and Software - ERTS2 2014*, pages 1–4, 2014.

[9] J. Boudjadar, J. H. Kim, and S. Nadjm-Tehrani. Performance-aware scheduling of multicore time-critical systems. In *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE*, pages 105–114. IEEE, 2016.

[10] F. Cassez and K. G. Larsen. The impressive power of stopwatches. In *CONCUR'00*, volume 1877 of *LNCS*, pages 138–152, 2000.

[11] X. Chen, Z. Xu, H. Kim, P. V. Gratz, J. Hu, M. Kishinevsky, U. Ogras, and R. Ayoub. Dynamic voltage and frequency scaling for shared resources in multicore processor designs. In *50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–7, 2013.

[12] Y. L. Chen, M. F. Chang, and W. Y. Liang. Dynamic voltage and frequency scaling based parallel scheduling scheme for video recognition on multicore systems. In *IEEE International Conference on Consumer Electronics-Taiwan (ICCE-TW)*, pages 1–2, 2016.

[13] A. K. Datta and R. Patel. Cpu scheduling for power/energy management on multicore processors using cache miss and context switch data. *IEEE Transactions on Parallel and Distributed Systems*, 25(5):1190–1199, May 2014.

[14] A. David, K. G. Larsen, A. Legay, M. Mikucionis, D. B. Poulsen, J. van Vliet, and Z. Wang. Statistical model checking for networks of priced timed automata. In *FORMATS*, volume 6919 of *LNCS*, pages 80–96. Springer, 2011.

[15] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Syst*, 17(2-3):131–181, 1999.

[16] P. Huyck. Arinc 653 and multi-core microprocessors; considerations and potential impacts. In *DASC'12*, pages 6B4–1–6B4–7, 2012.

[17] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 347–358. IEEE Computer Society, 2006.

[18] A. Kandhalu, J. Kim, K. Lakshmanan, and R. Rajkumar. Energy-aware partitioned fixed-priority scheduling for chip multi-processors. In *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, volume 1, pages 93–102, 2011.

[19] H. Karray, M. Paulitsch, B. Koppenhoefer, and D. Geiger. Design and implementation of a degraded vision landing aid application on a multicore processor architecture for safety-critical application. In *16th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC*, pages 1–8. IEEE Computer Society, 2013.

[20] B. Kim, L. Feng, L. T. X. Phan, O. Sokolsky, and I. Lee. Platform-specific timing verification framework in model-based implementation. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 235–240, 2015.

[21] H. Kim, D. de Niz, B. Andersson, M. H. Klein, O. Mutlu, and R. Rajkumar. Bounding memory interference delay in COTS-based multi-core systems. In *RTAS'14*, pages 145–154, 2014.

[22] H. Kim, A. Kandhalu, and R. Rajkumar. A coordinated approach for practical os-level cache management in multi-core real-time systems. In *ECRTS13*, pages 80–89, 2013.

[23] J. H. Kim, A. Boudjadar, U. Nyman, M. Mikucionis, K. G. Larsen, and I. Lee. Quantitative schedulability analysis of continuous probability tasks in a hierarchical context. In *2015 18th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE)*, pages 91–100, 2015.

[24] Y. Li, J. Niu, M. Atiquzzaman, and X. Long. Energy-aware scheduling on heterogeneous multi-core systems with guaranteed probability. *Journal of Parallel and Distributed Computing*, 103:64 – 76, 2017. Special Issue on Scalable Cyber-Physical Systems.

[25] C. C. Lin, C. J. Chang, Y. C. Syu, J. J. Wu, P. Liu, P. W. Cheng, and W. T. Hsu. An energy-efficient task scheduler for multi-core platforms with per-core dvfs based on task characteristics. In *43rd International Conference on Parallel Processing*, pages 381–390, 2014.

[26] A. Löfwenmark and S. Nadjm-Tehrani. Challenges in future avionic systems on multi-core platforms. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, pages 115–119, 2014.

[27] J. Lu and Y. Guo. Energy-aware fixed-priority multi-core scheduling for real-time systems. In *2011 IEEE 17th International Conference on Embedded and Real-Time Computing Systems and Applications*, volume 1, pages 277–281, 2011.

[28] J. Nowotsch, M. Paulitsch, D. Buhler, H. Theiling, S. Wegener, and M. Schmidt. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *Proceedings of ECRTS'14*, pages 109–118, 2014.

[29] C. Poellabauer, T. Zhang, S. Pande, and K. Schwan. An efficient frequency scaling approach for energy-aware embedded real-time systems. In M. Beigl and P. Lukowicz, editors, *Systems Aspects in Organic and Pervasive Computing - ARCS 2005: 18th International Conference on Architecture of Computing Systems, 2005.*, pages 207–221, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[30] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *ISCA '00*, pages 128–138. ACM, 2000.

[31] RTCA. DO-297/ED-124 - Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations, 2005.

[32] S. R. Sarangi, B. Greskamp, and J. Torrellas. Cadre: Cycle-accurate deterministic replay for hardware debugging. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 301–312, Washington, DC, USA, 2006. IEEE Computer Society.

[33] L. Subramanian, V. Seshadri, Y. Kim, B. Jaiyen, and O. Mutlu. Mise: Providing performance predictability and improving fairness in shared main memory systems. In *HPCA'13*, pages 639–650, 2013.

[34] J. R. Tramm and A. R. Siegel. Memory bottlenecks and memory contention in multi-core monte carlo transport codes. *Annals of Nuclear Energy*, 82:195 – 202, 2015. Joint International Conference on Supercomputing in Nuclear Applications and Monte Carlo 2013, SNA + MC 2013. Pluri- and Trans-disciplinarity, Towards New Modeling and Numerical Simulation Paradigms.

[35] G. Yao, R. Pellizzoni, S. Bak, H. Yun, and M. Caccamo. Global real-time memory-centric scheduling for multicore systems. *IEEE Trans. Computers*, 65(9):2739–2751, 2016.

[36] Y. Ye, R. West, Z. Cheng, and Y. Li. Coloris: A dynamic cache partitioning system using page coloring. In *Proceedings of PACT '14*, pages 381–392, 2014.