Energy-aware Real-Time Task Decomposition for partitioned-EDF Scheduling on Multi-core Uniform Architectures

Houssam Eddine, Richard Olejnik, El Hassen Benyamina, Giuseppe Lipari

Energy-aware Real-Time Task Decomposition for partionned-EDF Scheduling on Multi-core Uniform Architectures

Houssam Eddine ZAHAF[a,b], Richard OLEJNIK[a], Abou ElHassen BENYAMINA[b], Giuseppe LIPARI[a]

[a] *Lille University, France*
[b] *Oran 1 University, Algeria*

Abstract

Modern multi-core embedded processors allow to implement increasingly complex processing applications, as any processing of large amount of data. Many of these are submitted to timing constraints, and can be implemented as parallel tasks by taking advantage of a multi-core architecture. However a careless implementation can waste precious energy, especially when processing elements are powered by batteries. Thus, It is necessary to calibrate the operational frequency and voltage of the processors so that the consumed energy is reduced while still meeting the timing requirements.

In this paper, we explore the design choices in implementing parallel real-time applications on multi-core systems. We propose a realistic parallel real-time task model, NLP formulation and some heuristics for allocating threads to processors and selecting in offline their operational frequency. Experimental results with synthetic tasks sets show that it is time consuming to use NLP-solvers and that our heuristics are effective in reducing the power consumption.

Keywords: *Power*, *Real-Time*, *Parallel*, *Frequency*, Periodic tasks, Partitionned, Decomposition.

1. Introduction

Power consumption is an increasing concern in *cyber-physical* real-time systems, especially when processing elements operate on battery power. Embedded

real-time distributed systems must support increasingly complex applications such as distributed video surveillance, collection and analysis of a large amount of sensor data, etc.

To reduce the network traffic, instead of collecting and sending all data to a remote central server, a part of the processing is done on-site with medium-size embedded computing boards, so that only a small part of preprocessed data needs to be sent. For example, instead of sending the full video frames, a surveillance system analyses them on-site by extracting the important features, and sends only the features for further processing in the central site. This model of computation is now known as "Fog-computing" [1].

The embedded platforms used for supporting fog computing are often multicore systems and many of these applications can be easily parallelised by distributing data across the parallel computing elements. Reducing power consumption in these systems is a very serious problem when they are operated by batteries. Even when they are connected to the electric grid, we need to keep the consumption as low as it is possible. Multicore technology can help us in achieving timeliness and low power consumption: in fact, even when the computational load is not very high, multicore processors are more energy efficient than an equivalent single-core platform [2]. The idea is to operate the system at a lower frequency and at the same time reduce the response time by decomposing the tasks into parallel threads to be partitioned across the cores.

Task decomposition is a well-known problem in the parallel programming community. For example, OpenMP [3] is an API specification for parallel programming. The section of code that is meant to run in parallel is marked with a pre-compiler directive. One example of such directive is the *pragma parallel for*. In the OpenMP parallel model, a parallel *for* loop preceded by the STATIC schedule clause is implemented by distributing the N iteration on p threads into approximately $\frac{N}{p}$ iterations per each thread (if no further specification is implied). After the execution of the parallelized code, the threads join back into the master thread, which continues onward to the end of the program. However, this task decomposition does not take into account the task scheduling,

the respect of the real-time and energy constraints and the processor frequency definition. In this work we adress these problems.

Goossens et al. in [4] classified parallel real-time task models according to the way the level of parallelism is specified and to the moment of this specification. We can distinguish three types of models:

- *rigid*: the number of processors assigned to the task is specified independently before the scheduling analysis and does never change;
- *moldable*: the number of processors assigned to the task is specified off-line using an *off-line analysis algorithm*, for example during pre-processing or compilation and does not change after;
- *malleable*: the number of processors assigned to the task may dynamically change during execution.

According to [5, 6], most parallel applications in the real world are moldable. Hence we decided to focus our work on moldable real-time tasks.

Contributions of this paper.. This work focuses on intra-task parallelism to reduce energy consumption while respecting timing constraints. We select in off-line the frequencies of the core-groups of a multi-core platforms, by means of schedulability analysis. We propose a parallel task model that accounts for decomposition overhead and different discrete operating frequencies. The problem of finding an optimal solution is very complex, so we propose heuristic algorithms to achieve a good compromise between precision of the solution and running-time of the analysis.

Paper structuring.. The rest of the paper is structured as follows. In Section 2, we discuss the related works. Section 3 is dedicated to describing the model of the system and the problem formulation. We present allocation and frequency definition algorithms in Section 5. The experimental results are discussed in Section 6. Finally, in Section 7 we present our conclusions.

2. Related work

In the real-time systems literature, a lot of works [7, 8, 9, 10, 11] focused on static and dynamic Voltage and Frequency scaling for uni-processor architectures, and recently some researchers proposed similar techniques for homogeneous multiprocessors architectures.

Bini et al. [7] proposed a semi-linear model for the variation of the execution of a task τ_i, which we use in our own task model (see Section 3). They also proposed an algorithm to set the minimum processor frequency on single processor architectures, so that all the deadlines are respected.

Concerning real-time parallel task scheduling on multi-processors, many papers (e.g. [12, 13, 14, 15, 16, 17, 18]) focused on the intra-task parallelism, without considering energy-aware scheduling.

Kato et al. [14] proposed the *gang task* model. A task is defined by the number of processors used simultaneously, its execution time and the deadline that must be lower than the period (constrainned deadline). All threads of a parallel section in a gang model have to be run in parallel (they must start and stop the execution on their processors simultaneously). However, this model is more difficult to implement, because the scheduler needs to synchronize the execution of the threads among different processors. A different model, easier to implement, is the *multi-thread* model, where parallel threads are scheduled independently, and in this work we will adopt this model. Lakshmanan et al. [12] proposed a *fork-join* model for representing parallel tasks: each task is a sequence of alternating parallel and sequential *segments*. Saifullah et al. [18] proposed a parallel task model where a task is composed of segments, and each segments consists of a set parallel threads with the same real time characteristics (e.g. release time, execution time, deadline). At the end of each segment, parallel threads must synchronize. They also proposed a method to adapt other task models to their model.

Courbin and Goossens [13] proposed a less restrictive model where parallel

sub-tasks of each *phase*[1] have independent real time characteristics. They state that the model proposed in [18] is a specific case of their model. They also proposed an algorithm to assign real time parameters to a fork-join model in order to adapt it to their model called *Multi-phase Multi-Thread*. These papers [13, 18, 12] only address the scheduling problem of parallel task, without considering the energy consumption problem.

Fisher et al. in [19] defined the *optimal* frequency for minimizing energy consumption on homogeneous platforms with *malleable* tasks (see Section 3). They target homogeneous processor platforms with the ability of turning off some processors. They considered sporadic implicit deadline tasks and use a canonical parallel scheduler proposed in [17], which is an optimal scheduling algorithm for sporadic tasks on homogeneous multiprocessor platforms. We propose a similar approach, but we apply it on uniform multiprocessors and a different task model.

3. System Model

3.1. Task Model

Let $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ be a set of periodic moldable independent tasks. Every task τ_i is characterised by the tuple $\tau_i = (\mathsf{O}_i, \mathsf{D}_i, \mathsf{T}_i, \Gamma_i)$, where:

- O_i is the offset. It represents the release time of the first instance of the task τ_i. If $\forall i, O_i = 0$, then the task set is *synchronous*.
- D_i is the task's relative deadline, we consider tasks with D_i less than T_i.
- T_i: is the task's period, that means the time between the releases of two consecutive instances of task τ_i.
- Γ_i is the task's set of k *cut-points*, $\Gamma_i = \{\gamma_{i,1}, \ldots, \gamma_{i,K_i}\}$. Each cut-point $\gamma_{i,k} = \{\mathsf{Th}_{i,k,1}, \mathsf{Th}_{i,k,2}, \ldots\}$ is itself a list of parallel threads, and it rep-

[1] A phase is equivalent to a *segment*

resents one possible parallel decomposition of the task τ_i into parallel threads. We denote as $|\gamma_{i,k}|$ the number of threads in the cut-point.

Each thread $\mathsf{Th}_{i,k,z}$ has the same period and the same relative deadline of the task to which it belongs. In this work, we assume that the worst case execution time ($\mathcal{C}$) of every thread depends on the operating frequency of the processor on which it has been allocated: $\mathcal{C}_{i,k,z}(f_p) = \mathsf{ct}_{i,k,z}(f_p) + \mathsf{mt}_{i,k,z}$, where $\mathsf{ct}_{i,k,z}(f_p)$ is the part of the computation time that depends on the processor frequency and $\mathsf{mt}_{i,k,z}$ is the part that does not depend on the processor frequency (e.g. due to central memory accesses), So each thread is defined $\mathsf{Th}(\mathsf{ct}, \mathsf{mt})$. This model is similar to the one proposed by Bini et al in [7]. In particular, we use a semi-linear model:

$$\mathcal{C}_{i,k,z}(f_p) = \frac{\mathsf{ct}_{i,k,z}}{s_p} + \mathsf{mt}_{i,k,z} \tag{1}$$

where $s_p = f_p/f_{max}$ is the *core speed* (or *relative frequency*) of the core where the task is executing. We denote by $\mathcal{C}_{i,k} = \sum_z \mathcal{C}_{i,k,z}(f_p)$ the sum of the execution times of the threads in the cut-point $\gamma_{i,k}$. For every possible combination of operating frequencies, we assume that $\mathcal{C}_{i,k}$ is monotonic non decreasing with the number of threads in the cut-point. In other words, using more threads implies a larger overhead, due to several factors: synchronisation, caches, conflicts in accessing the memory bus. Under this assumption, it does not make sense to have $|\gamma_{i,k}| > m$.

For example, consider task $\tau_1 = (\mathsf{O}_1 = 0, \mathsf{D}_i = 11, \mathsf{T}_i = 15, \Gamma_1)$, Γ_1 consists of 3 cut-points (see Fig. 1):

- $\gamma_{1,1} = \{\mathsf{Th}_{1,1,1}(11, 6)\}$,
- $\gamma_{1,2}\{\mathsf{Th}_{1,2,1}(6, 3); \mathsf{Th}_{1,2,2}(6, 4)\}$,
- $\gamma_{1,3} = \{\mathsf{Th}_{1,3,1}(2, 4); \mathsf{Th}_{1,3,2}(6, 2); \mathsf{Th}_{1,3,3}(3, 3)\}$

Assuming $s_p = 1$ for all processors, this task can be allocated on three different forms: one thread with the execution time $\mathcal{C}_{1,1} = 11 + 6 = 17$ using the first cut-points $\gamma_{1,1}$; two parallel threads with the execution time $\mathcal{C}_{1,2,1} = 9$,

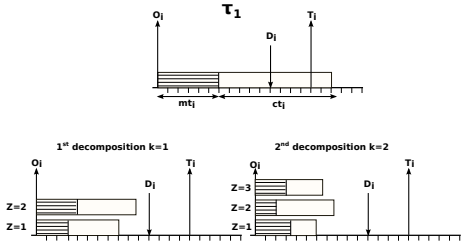

Figure 1: Task decomposition

$\mathcal{C}_{1,2,2} = 12$ using the second cut-point $\gamma_{1,2}$; 3 parallel threads with the execution time $\mathcal{C}_{1,3,1} = 6$, $\mathcal{C}_{1,3,2} = 8$, $\mathcal{C}_{1,3,3} = 6$ using the third cut-point $\gamma_{1,3}$.

Notice that we allow different values of ct and mt among parallel threads belonging to the same cut-point; and we allow many cut points with the same number of threads. Thus, this model is quite general and can be used to represent alternative decompositions for the same level of parallelism.

3.2. Uniform architectures and Power Model

In this paper we address uniform multi-core architectures. In this kind of architectures, cores have the same instruction set, and they may differ on their operating frequencies. We assume that the architecture supports voltage and frequency scaling technology, which allows tasks to run at different frequencies/speeds, and that each core has a discrete set of operating frequencies.

The execution time of a task does not depend only on the core speed. Seth et al. [20] noticed that not all the task execution times scale on the core speed (i.e. frequency) because some part of the code deals with the memory and I/O devices.

In order to validate these assumptions, we implemented a simple benchmark consisting of the multiplication of two matrices of 300×300. We implemented the task and run it on a Linux OS running on a *i3-3217U* processor from *Intel*. The system has 3Mb of cache memory, two physical cores, each one hosting 2 logical cores by using hyper-threading. The frequency of each physical core can be set independently from 800 MHz to 1800 MHz. The task is implemented in

C with using directly the Pthread library and it is decomposed into a set of n threads, with n varying between $[1, 2, 4, 6, 8]$. Each thread process one portion of the matrix, and the decomposition is fair: all threads process approximately $300/n$ lines. The task is periodic with a period of $T = 200$ msec. Finally, other tasks in the system run with non-real-time priority trying to load the cache and the sytem.

Figure (2) shows the average execution time of each thread when all cores operate at a frequency of $1,800$MHz. Each point represent the average execution time of a decomposition from 1 to 8 threads.

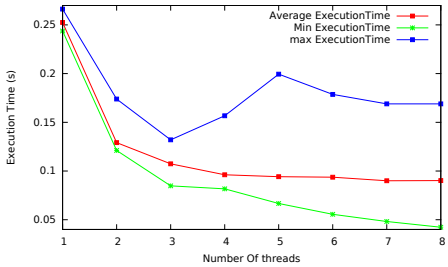

Figure 2: Task decomposition

Figure (3) shows the average execution time of the same task decomposed into $1, 2, 4$ threads at different operating frequencies. We use this experiment to estimate the values of ct and mt for the different threads in the decomposition.

N. Threads	Parameters
1	(mt = 60ms, ct = 490ms)
2	(mt = 33ms, ct = 258ms)
4	(mt = 13ms, ct = 185ms)

We also plotted the results of the execution time obtained by Equation (1) for the case of one thread (blue line). Our model is pretty close to the real execution time (red line), whereas a simple linear model (green line) may produce an underestimation. Similar results are obtained with a different decomposition[2].

[2]Not shown here, to avoid cluttering the figure with too many lines.

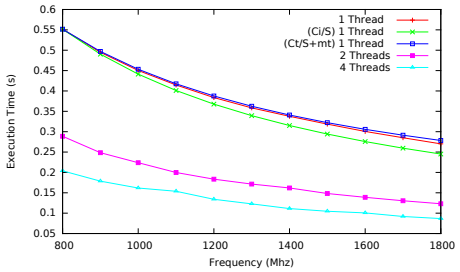

Figure 3: Task decomposition

We model the hardware architecture as a list of L groups of cores. Cores from the same group ($l_d, d \in \{0 \dots L\}$) operate at the same frequency. For example, for the Intel architecture described above, we have 2 physical cores, each one supporting two logical cores. Therefore, we set $m = 4$, and $L = 2$. Logical cores in the same group reside on the same physical core, so they must have the same frequency. If $m = L$ then each core can change the its frequency independently. A processor can be put in a deep low-power mode state, so it is turned off from the point of view of executing tasks, even if it can still consume some small amount of energy. This model allows us to address many different uniform architectures like the *ix* Intel family and the ARM big.LITTLE architecture [21].

3.3. Energy Consumption

In CMOS circuits, we distinguish between *dynamic power consumption*, that is the power dissipation due to switching, and *static power consumption*, mostly due to leakage current dissipated in electronic components like capacitance and transistors.

Jing Mie et al. [22] defined the dynamic power dissipated by a CMOS circuit as the product of a constant coefficient that depends on the technology used ξ, by the square of the voltage, and by the frequency.

$$E = \xi \times V^2 \times f \tag{2}$$

They also defined the frequency as the ratio of the difference between the actual

voltage V and V_{Th} raised to the power of a, where V_{th} is the threshold voltage by the product of a constant K and the logic depth L_d. a and K are constants that depend on the technology.

$$f = \frac{(V - V_{Th})^a}{K \times L_d} \tag{3}$$

By combining the two equations, the consumed energy E can be expressed as the product of a constant *Const* by the frequency f power λ (4). We will use this equation to compute the dissipated energy in Section 4.

$$E(f) = Const \times f^{\lambda} \tag{4}$$

3.4. Scheduling

In this paper we assume a partitioned Earliest Deadline First (EDF) scheduling. Each core has its own single-processor EDF scheduler and a separate ready-queue. Therefore the scheduling analysis can be performed with well-know techniques for single-processor scheduling. We assume that all tasks are independent of each other, whereas threads belonging to the same tasks and running on different cores need to synchronize their activation times and deadlines.

The scheduling analysis is based on the well know *demand-based analysis*. Let $\mathcal{T}_p$ denote a task set of n periodic tasks allocated on the same core, $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$, and let t be a non-negative integer. The *Demand bound function* $\mathsf{dbf}(\mathcal{T}, t)$ denotes the maximum cumulative execution requirement that could be generated by jobs of $\mathcal{T}_p$ that have both ready times and deadlines within any time interval of duration $[0 - t]$. Equation (5) describes the formula for computing the dbf:

$$\mathsf{dbf}(\mathcal{T}_p, t) = \sum_{\mathsf{Th}_i \in \mathcal{T}_p} \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor C_i \tag{5}$$

Notice that C_i denotes generically the computation time of the thread, and of course it depends on the frequency at which the core is operated, as described in Equation (1).

To be schedulable under EDF, the dbf value of a task set $\mathcal{T}_p$ at each moment of time t_0, must be equal or less than t_0 (Equation 6).

$$\forall t_0 \leq t^*, \mathsf{dbf}(\mathcal{T}_p, t_0) \leq t_0 \tag{6}$$

where t^* is a constant that depends on the utilisation of the task set (see [23] for more details on the analysis algorithm).

4. Problem formulation

In order to compare our scheduling and frequency heuristics to optimal ones, we formulate our problem as a non linear optimisation problem, We use the the follow acronyms in problem definition:

- L : it the number of core groups;
- h: is the hyper period of tasks, and it is computed as least common multiple of all periods of tasks;
- f_l: is the operating frequency of a core in group l;
- s_l: is the speed of a core in group l;
- $\mathsf{dbf}_{i,k,z}(s_p, t)$: is the value of the demand bound function of thread $\mathsf{Th}_{i,k,z}$ when allocated on a core p with speed s_p at time t.

Let $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ a task set of n periodic tasks on a uniform architecture of m cores. We define the binary variable $x_{i,k,z,p}$ as:

$$\begin{cases} 1 & \text{if thread } \mathsf{Th}_{i,k,z} \text{ is allocated on core } p \\ 0 & \text{otherwise} \end{cases}$$

We assume that the cores of each group are indexed alternatively with the next groups. For example, in an architecture of $m = 12$ cores with three groups, the cores of group l_0 are indexed as $0, 3, 6, 9$, those of l_1 are indexed as $1, 4, 7, 10$, and those of l_2 are indexed $2, 5, 8, 11$. Thus, the group of core p is $l_p = p\%L$.

Given these definitions, we now present one formulation of the problem as a Non-Linear Programming problem:

$$min\ E = \xi \sum_{i=1}^{n} \sum_{p=1}^{m} \sum_{k=1}^{K} \sum_{z=1}^{z_k} f_{l_p}^3 \times (\frac{\mathsf{ct}_{ikz}}{s_{l_p}} + \mathsf{mt}_{ikz}) \times x_{i,k,z,p} \tag{7}$$

subject to the following constraints:

$$\sum_{j=1}^{m} \sum_{k=1}^{K} \sum_{z=1}^{m} x_{i,k,z,p} = m \quad \forall i \tag{8}$$

$$\sum_{i=1}^{n} \mathsf{dbf}_{i,k,z}(s_p, t) x_{i,k,z,p} \leq t \quad \forall p, t \in [0 - h] \tag{9}$$

$$x_{i,k,z,p} + x_{i,k',z',p'} \leq 1, \quad \forall i, k, z, p, k' > k, z', p' \tag{10}$$

$$\sum_{j=1}^{m} x_{i,k,z,p} \leq 1, \quad \forall i, k, z \tag{11}$$

To grant the respect of the schedulability of each task set $\mathcal{T}_p$ on core p at each time t, the dbf must verify Condition (9). Constraint (8) expresses that all threads of the same cut point must be scheduled. A correct solution of the problem must verify that all the threads of the same task belong to the same *cut-point*. We express this as a conflict constraint which mean that whenever a cut-point is choosen, all the other cut-points are ignored. (Constraint (10)). Finally, we must verify that each thread is allocated on only one core (Constraint (11)).

There are many NLP solvers, both freely available as open source software, and commercial ones. We used CPLEX [24] and Knitro [25] solvers with academic licences. Unfortunately the problem at hand is a very complicated combinatorial problem, and a small example with 3 task with 5 cut-points per each task, to be executed on 4 cores with 10 frequency levels, takes several hours. We believe that it is impractical to use a NLP solver for this kind of problems. Thus, in this paper we propose heuristics to obtain quasi-optimal solutions in a reasonable time. In the next section we present our heuristic algorithms.

5. Partitioning

In this section we propose some heuristics for solving the problem approximately in a reasonable time. Our algorithm:

- selects the level of decomposition for each task by selecting the cut-points;
- selects the frequency for each core group;
- assigns threads to cores;

so to minimise energy consumption while guaranteeing that all deadlines are respected.

Let $\mathcal{T} = \{\tau_1, \tau_1, \ldots, \tau_n\}$ be a set of n tasks. We start by defining the *baseline utilisation* for each task as $U_i = \frac{\mathcal{C}_i}{T_i}$, where $\mathcal{C}_i$ is the WCET of the single-thread cut-point, considering a processor of speed 1. Then $U = \sum_i U_i$ is the baseline utilisation of the system.

We also define $\mathcal{S}$ the *cumulative strength* of the our architecture, that is the sum of all processors speed: $\mathcal{S} = \sum_{j=1}^{m} s_j$). To be schedulable, the following necessary condition must hold:

$$U \leq \sum_{p=1}^{m} s_{l_p} \tag{12}$$

We will use a classical "greedy" approach. We start by analysing frequency configuration having the minimal strength that verifies Equation (12), then we do partitioning and we check the schedulability. If no feasible solution can be found, we keep increasing the strength until we find a feasible partitioning.

Before explaining our algorithm in more details, we need to introduce some more definitions.

- m_l is the number of cores in group l;
- act_l is the number of active cores in group l
- sl_l is the lower bound speed of the cores in group l
- sh_l is the upper bound speed of the cores in group l

- so_l is the operating speed of group l
- $\mathcal{S}_l$ is the cumulative strength of group l, $\mathcal{S}_l = m_l s_l$

Algorithm 1 Full algorithm

Input: $\mathcal{T} : TaskSet, L : Int, U : Float, m, \mathsf{sl}, \mathsf{sh} : Array[Int]$;
$\mathcal{S}^{max} = \sum_{l=0}^{L} \mathsf{sh}_l * m_l$
$U = \sum_{i=1}^{n} \frac{\mathsf{ct}_i + \mathsf{mt}_i}{T_i}$
$\mathcal{S}^{temp} = U$
while $\mathcal{S}^{temp} \leq \mathcal{S}^{max}$ **do** ▷ main loop
  arch = selectFrequencies(L, $\mathcal{S}^{temp}$, m, sl, sh)
  if isCPSchedulable(Arch, $\mathcal{T}$) **then**
    return true;
  else
    $\mathcal{S}^{temp} = \mathcal{S}^{temp} + \mathsf{Step}$
  end if
end while
return false;

Algorithm (1) increases the cumulative strength $\mathcal{S}^{temp}$ at each iteration by Step, then it selects the frequencies of all groups by invoking Algorithm 2, and finally it and allocates the threads and checks schedulability by invoking Algorithm 5. It returns true as soon as it finds a feasible solution. In the next subsection we will explain the algorithm for assigning the frequencies and allocating the tasks.

5.1. Frequency selection

The goal of this step is to select the frequency of each core group, so that the cumulative strength of the platform equals to $\mathcal{S}^{temp}$. We need also to respect the frequency bounds for each group. We assume that groups are ordered by increasing maximal frequency, so that less powerful groups are loaded by threads before higher frequency groups.

The heuristic returns the operating frequency so_l and the number of active cores act_l of each group l. As anticipated before, Equation (12) must be verified. Let us rewrite this equation by replacing $\sum_{p=1}^{m} s_{l_p}$ by the equivalent value $\sum_{l=1}^{L} \mathcal{S}_l$:

$$U - \mathcal{S}_0 - \mathcal{S}_1 \ldots - \mathcal{S}_L \leq 0$$

Let $\mathsf{ex}_0 = U - \mathcal{S}_0$ be the *excess bandwidth* for group $l = 0$. For the following groups, $\mathsf{ex}_l = \mathsf{ex}_{l-1} - \mathcal{S}_l$. By substituting, we get:

$$\forall l \geq 0, ex_l - \mathcal{S}_{l+1}, - \ldots - \mathcal{S}_L \leq 0$$

For all groups with $\mathsf{ex}_l > 0$, we just turn on all cores and set them at the maximum speed: $\mathsf{so}_l = \mathsf{sh}_l$ and $\mathsf{act}_l = m_l$.

Suppose that $ex_l \leq 0$ for some group l. It means that the cumulative strength up to group l can support the current load. Then we set all successive strengths $\mathcal{S}_{l'}, l' > l$ to 0 by turning off all cores in the groups. In addition, we distribute the current remaining bandwidth equally on all cores of group l:

$$\mathsf{so}_l = round - tomode(\frac{\mathsf{ex}_{l-1}}{m_l}) \tag{13}$$

If the computed speed so_l is greater than the lower bound speed of the current group l (see Equation (**??**)), the number of active cores is set equal to the number of cores in the group:

$$\mathsf{act}_l = m_l \tag{14}$$

Otherwise, the frequency is set to the minimal frequency sl and the number of active cores is set to the ratio between the ex_{l-1} and the operating frequency at the current group so_l.

$$\mathsf{act}_l = \lceil \frac{\mathsf{ex}_{l-1}}{\mathsf{so}_l} \rceil \tag{15}$$

In this case, the assigned speed at this level may be greater than the remaining strength. We compute this overhead as:

$$\rho = \mathsf{so}_l * \mathsf{act}_l - \mathsf{ex}_{l-1} \tag{16}$$

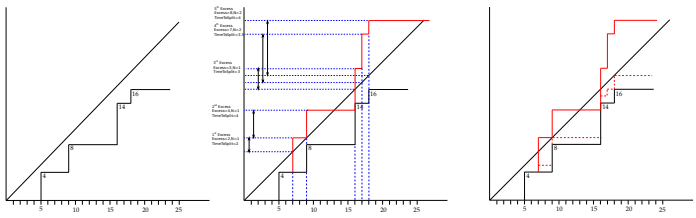

Figure 4: Decomposition example

and we remove it from cores speed of a lower group:

$$\mathsf{so}_{l-1} = \mathsf{so}_{l-1} - \frac{\rho}{\mathsf{act}_{l-1}} \tag{17}$$

The complexity of the setting frequencies Algorithm 2 is trivially $\Theta(L)$. The method *round-to-mode* selects the first higher possible frequency, since the frequencies are discrete.

5.2. CP partitionning

After selecting the frequencies, we proceed to decomposition and allocation. We consider partitioned-EDF scheduling, and we propose an iterative heuristic for solving the task decomposition and allocation problem at once. At each iteration, we check if the task fits entirely in the current core or if it needs to be split. In the latter case, the heuristic defines the part (threads) that will be partitioned on the current core, and the part (threads) that will be allocated in the next iteration.

We preliminary sort all cores in non-increasing order according to their operating speed, then we go trought three steps. First, we select the task τ at the top of the task list, and the core with the highest index (hence the lower speed), let it be core p. Then the dbf is computed for the set of threads already allocated on the core p, together with the new task τ for the speed s_p:

$$\mathsf{dbf}(\mathcal{T} \cup \{\tau\}, s_p, t) = \mathsf{dbf}(\mathcal{T}, s_p, t) + \mathsf{dbf}(\{\tau\}, s_p, t) \tag{18}$$

1. if $\forall t,\ t \in [0, t^*]$ Equation (19) is verified

$$\mathsf{dbf}(\mathcal{T} \cup \{\tau\}, s_p, t) \leq t, \tag{19}$$

Algorithm 2 selectFrequencies

Input: L:Int, $\mathcal{S}^{temp}$: Float, m,sl, sh: Array[Int];

Output so: Array[Float], n: Array[Int];

ex: Array[Float](L+1);

$\mathsf{ex}_0 = \mathcal{S}^{temp}$

$\mathcal{S}$:Float=0

for $(l = 0\ to\ L-1)$ **do**

$\mathcal{S} += m_l \times \mathsf{sh}_l$

end for

if $(\mathcal{S} < \mathcal{S}^{temp})$ **then**

return false;

end if

for $(l = 0\ to\ L-1)$ **do**

$\mathsf{ex}_{l+1} = \mathsf{ex}_l - \mathsf{sh}_l \times m_l$

if $(\mathsf{ex}_{l+1} \leq 0)$ **then**

$\mathsf{so}_l = \textit{round-to-mode}(\frac{ex_{l+1}}{m_l})$

if $(\mathsf{so}_l < \mathsf{sl}_l)$ **then**

$\mathsf{so}_l = \mathsf{sl}_l$

$\mathsf{act}_l = \lceil \frac{ex_{l+1}}{\mathsf{so}_i} \rceil$

$\rho = \mathsf{so}_l \times \mathsf{act}_l - \mathsf{ex}_{l+1}$

$\mathsf{so}_{l-1} = \textit{round-to-mode}(\mathsf{so}_{l-1} - \frac{\rho}{\mathsf{act}_{l-1}}))$

else

$\mathsf{act}_l = m_l$

break loop

end if

else

$\mathsf{so}_l = \mathsf{sh}_l$

$\mathsf{act}_l = m_l$

end if

end for

the task set $\mathcal{T}_p \cup \{\tau\}$ is schedulable, hence task τ is allocated on the core p.

2. if, for some t values, Equation (19) is not verified, then the task set $\mathcal{T} \cup \{\tau\}$ is not schedulable. In this case, we seek to split the reduce the execution time of task τ so that one part will fit between the $\mathsf{dbf}$and t so that Equation 19 is verified. We call the time to be reduced *excess-time*. To evaluate the excess-time, first we evaluate the cumulative excess of all instances on the task form 0 to t:

$$\delta = \mathsf{dbf}(\mathcal{T}_p \cup \{\tau\}, s_p, t) - t \tag{20}$$

so the *excess-time* by one instance is equal to the ratio of the δ and the number of instances of the task τ in the interval $[0, t]$:

$$\begin{cases} \delta = \mathsf{dbf}(\mathcal{T}_j, j, t) - t \\ \#instances = \dfrac{t}{T_i} \\ excess\text{-}time(t) = \lceil \dfrac{\delta}{\#instances} \rceil \end{cases} \tag{21}$$

For all t , where condition 19 is not verified, *excess-time* is evaluated and only the maximal *excess-time* is maintained. The evaluation of *excess-time* is described in Algorithm 3. The complexity of this algorithm is $\Theta(h \times \sum_{i=1}^{n} max(\mid \gamma_{i,k} \mid))$.

Example. In the example of Figure 4, we try to evaluate the *excess-time* for the task $\tau(O = 0, D = 7, T = 10, \{ct = 8, mt = 1\})$ on a core of speed $s = 0.5$. We assume that the task set $\mathcal{T}$ is already allocated on this core and the value of the $\mathsf{dbf}(\mathcal{T}, s_p, t),\ t \in [0, 25]$ is presented on the first graph of Figure 4. The second graph represents the dbf of the new task set $\{\mathcal{T} \cup \tau\}$ and the *excess-time* points, which are reported, also, in table below.

Algorithm 3 *excess-time*Evaluation

Input: Taskset:$\mathcal{T}$, core: P, Task :τ

Output: *excess-time* = 0

Temp-excess:Integer = -1

$t = 0$

while $(t \leq h)$ **do**

$\mathsf{dbf}^* = \mathsf{dbf}(\mathcal{T}, s_p, t) + \lfloor \frac{t+(T_i-D_i)}{T_i} \rfloor \frac{\mathsf{ct}_i}{s_p} + \mathsf{mt}_i$

if ($\mathsf{dbf}^* > t$) **then**

$\delta = \mathsf{dbf}^* - t$

$n = \lceil \frac{t}{T} \rceil$

Temp-excess = $\lceil \frac{\delta}{n} \rceil$

if (Temp-excess > *excess-time*) **then**

T*excess-times* = Temp-excess

end if

end if

t=t+1

end while

Time	Excess	Nbr jobs	ToCut
7	2	1	2
9	4	1	4
16	3	1	3
17	7	2	3.5
18	8	2	4

The value of the *excess-time* corresponds to the maximum value among all excesses (*excess-times* = 4, third graph of Figure 4).

Now that we have introduced the concept of excess-time, we use a simple heuristic to decompose the tasks that do not fit in a core. If *excess-time* $\geq \frac{\mathsf{ct}_i}{s_p} + \mathsf{mt}_i$, (this means that the whole task is in excess), we seek to allocate the selected task on the next core $p+1$. If all cores are already investigated, the task set is not schedulable.

If *excess-time* $\leq \frac{\mathsf{ct}_i}{s_p} + \mathsf{mt}_i$, the task τ_i must be split according to one possible $\gamma_{i,k} \in \Gamma_i$. We use a search function that returns two lists: the first one consists of the threads that fit on the current core, whereas the second list of threads is put back into the task list to be allocated in the next iterations.

Let $\gamma_{i,k}$ be a *cut-point* of Γ_i of the task τ_i. The $\gamma_{i,k}$ is as a corret decomposition according to *excess-time* if and only if Equation 22 is verified:

$$\begin{aligned} &\exists \sigma_{i,k} \subset \gamma_{i,k}, \quad \sigma_{i,k} \neq \emptyset, \gamma_{i,k} \neq \sigma_{i,k} \\ &\sum_{Th_{i,k,z} \in \sigma_{i,k}} C_{i,k,z} \geq \textit{excess-time} \end{aligned} \tag{22}$$

where $\sigma_{i,k}$ is the list of threads to put back to the task list, and $\gamma_{i,k} \setminus \sigma_{i,k}$ is the list of threads to be allocated to the current core. So, For the same excess-time, we can have numerous correct cut-points.

For example, let *excess-time* = 5 and $\Gamma_1 = \{\gamma_{1,1} = \{\mathsf{Th}_{1,1,1}(\mathsf{mt} = 2, \mathsf{ct} = 5), \mathsf{Th}_{1,1,2}(\mathsf{mt} = 2, \mathsf{ct} = 5),\}, \gamma_{1,2} = \{\mathsf{Th}_{1,2,1}(\mathsf{mt} = 1, \mathsf{ct} = 2), \mathsf{Th}_{1,2,2}(\mathsf{mt} = 1, \mathsf{ct} = 2), \mathsf{Th}_{1,2,3}(\mathsf{mt} = 1, \mathsf{ct} = 5),\}\}$. This task can be split in several threads that respect the *excess-time* as shown in the following table:

$\gamma_{1,k}$	return to TS	maintained
$\gamma_{1,1}$	$\mathsf{Th}_{1,1,1}$	$\mathsf{Th}_{1,1,2}$
$\gamma_{1,1}$	$\mathsf{Th}_{1,1,2}$	$\mathsf{Th}_{1,1,1}$
$\gamma_{1,2}$	$\mathsf{Th}_{1,2,1}, \mathsf{Th}_{1,2,2}$	$\mathsf{Th}_{1,2,3}$
$\gamma_{1,2}$	$\mathsf{Th}_{1,2,1}, \mathsf{Th}_{1,2,3}$	$\mathsf{Th}_{1,2,2}$
$\gamma_{1,2}$	$\mathsf{Th}_{1,2,2}, \mathsf{Th}_{1,2,3}$	$\mathsf{Th}_{1,2,1}$
$\gamma_{1,2}$	$\mathsf{Th}_{1,2,3}$	$\mathsf{Th}_{1,2,1}, \mathsf{Th}_{1,2,2}$

Thus, the cut-point selection (Algorithm (4)) has a significant impact on scheduling. The algorithm of search for the cut-point is simple. First, it seeks the thread $\mathsf{Th}_{i,k,z}$ that has the exact values of the *excess-time*. If found, this *cut-point* will be returned. Otherwise, it looks for the closest combination of threads that is greater or equal to the *excess-time*.

The complexity of **lookForCP** search (algorithm 4) is $\Theta(max(\mid \Gamma_i \mid) \times max(\mid \gamma_{i,k} \mid))$. However, after splitting, the cut-points list Γ_i is reduced to one cut-point (constraint that all threads must be from the same cut point, Equation (22) $(\gamma_{i,k} \setminus \sigma)$). Thus, the **lookForCP** search function complexity is reduced considerably after the first allocation.

The partitioning heuristic is described in (Algorithm 5). The complexity of this algorithm is $\Theta(n \times m \times h \times \sum_{i=1}^{n} max(\mid \gamma_{i,k} \mid))$.

The full algorithm (1) combine all the heuristics described in this section. the complexity of this algorithm is $\Theta(nbrIt \times n \times m \times h \times \sum_{i=1}^{n} max(\mid \gamma_{i,k} \mid))$ where *nbrIt* is the number of iterations from the lowest cumulated strength to the highest cumulated strength.

6. Simulations and Results

In order to evaluate our contributions, we applied our algorithm to several synthetically generated scenarios. We selected an architecture consisting of $m = 4$ processors with $L = 2$ groups of 2 cores each. Each core has several operating frequency levels ranging from 800MHz to 1800MHz, which corresponds to speed levels ranging in between $[0.44, 1]$. This corresponds to a cumulative maximum

Algorithm 4 lookForCP

Input: *excess-time*,s: Float,τ_i: CPTask

Output:(List[ThreadCP], List[ThreadCP])

l: List[ThreadCP] = Nil;

lm: List[ThreadCP] = Nil;

lout: List[ThreadCP] = Nil;

$Kept = \mathsf{ct}_i + \mathsf{mt}_i - ex \times s$

for $(\forall \gamma_{i,k} \in \Gamma_i)$ **do**

cum:Float = 0

l = Nil

for $(\forall \mathsf{Th}_{i,k,z} \in \gamma_{i,k})$ **do**

if $(\lceil (ct_{i,k,z}/s) \rceil + \mathsf{mt}) \leq Kept)$ **then**

if $\lceil (\sum_{z=1}^{|\gamma_{i,k}|} \mathsf{ct}/s) \rceil + \sum_{z=1}^{|\gamma_{i,k}|} \mathsf{mt} - Kept > \lceil (ct_{i,k,z}/s) \rceil + \mathsf{mt}_{i,k,z}) - Kept) \mid$ **then**

lm = lm.add(cp)

lout = $\gamma_{i,k}$ - lm

end if

l = l.add(cp);

$cum+ = \lceil \mathsf{ct}_{i,k,z}/s) \rceil + \mathsf{mt}_{i,k,z}$

if (cum $\leq$ Kept) **then**

lm = l;

lout = $\gamma_{i,k}$ - lm;

end if

end if

end for

end for

return (lm, lout)

Algorithm 5 *cut-point* model partitioning

```
OrderTasks()
sortCoresBySpeed()
for τ_i ∈ AllTasks do
    for P_j ∈ AllCores do                                          ▷ loop2
        excess-time = excess-TimeEvaluation(T_{p_j}, s_j, τ);
        if ( excessT-time = 0 ) then                               ▷ dbf(T_j, s_j, t) ≤ t
            P.Allocate(τ)
            break loop2
        else
            if ( C_i^{s_j} ≤ excess-time ) then
                (σ, LM)=lookForCP(excess-time,s_j, τ_i)
                τ_1= createTask (maintained)
                τ_2= createTask (σ)
                P.Allocate(τ_1)
                AllTasks.add(τ_2)
                break loop2
            else
                SeekOnTheNextCore
            end if
        end if
    end for
end for
```

strength of $\mathcal{S} = 4$. These parameters correspond to the Intel i3-3217U used for the experiments shown in Section 3.2.

We generate utilization-rates between $[1 - 4]$, and the number of tasks between 2 and 5 by core.

6.1. Task Generation

The UUniFast method [26] can be used to generate a number of utilizations factors, whose sum is a given number bounded by 1 (i.e. $\forall i,\ u_i \leq 1$). The method has been extended to multi-core systems with the UUniFast-discard algorithm. The latter allows to set the bound to a number $m \geq 1$ and adds the constraint that each utilisation must not exceed 1.

In our case, we removed the latest constraint, because we allow tasks whose sequential utilisation is greater than 1.

In order to select the periods, we use the method proposed in [27] to bound the hyperperiod of the task set and generate task periods.

Therefore, we compute the computation times of the single-thread cut point for each task as $\mathcal{C}_i = u_i \times T_i$. Then, we generate

$$\mathsf{mt}_i = \mathcal{C}_i \times P \qquad \mathsf{ct}_i = \mathcal{C}_i - \mathsf{mt}_i$$

where P is the percentage of central memory access operations among all operations. We consider such percentage as a fixed number to reduce the number of parameters and simplify the experiments, but in reality such percentage depends on the structure of the task sets, the amount of data, the cache size, the interference from other tasks in the cache, etc.

Then, for each task we randomly generate the number of cut points i_K ($\mid \Gamma_i \mid$) between 2 and 5. For each cut point $\gamma_{i,k}$, we generate the number of threads Z_{i_k}. Finally, we use again UUniFast to generate the utilisation for each thread. To take into account the overhead of the decomposition (due to synchronization barriers, scheduling, etc.) we inflate the utilisation of the task to $U_i' = U_i \times (1 + \mathsf{cost} Z_k)$, where cost is a constant that represent the overhead in percentage. We fixed this overhead to $\mathsf{cost} = 0.05$. If the utilisation of a task

exceeds 1, $U_i \leq 1$, we generate the deadline between C_i and T_i. Otherwise, the deadline is generated in the interval $[P_r \times T_i, T_i]$ where P_r is generated between $[0.75 - 1]$. The task generation is described in Algorithm 6.

6.2. Power dissipation model

The goal of these experiments is to compute the relative performance of different algorithms. Therefore, just for the sake of this comparison, we consider $\xi = 1$. This is the same approach followed in Fisher et al. [19], and in Han et al [28]. So we consider the power consumption as $E(f) = f^3$. The dynamic energy spent by core p from $[t_1 - t_2]$ is simply $E = f_p^3 \times (t_2 - t_1)$. This allows to relatively compare the energy consumption for different scheduling algorithms but does not give an absolute power value.

6.3. Results and discussions

In this section we present and discuss the results of the experiments. Our task model is unique in that it allows parallel decomposition of a task in several different ways. No other algorithm can do the same, so we decided to compare against traditional allocation algorithms that do not decompose tasks.

In particular, the structure of a competing algorithm is the following:

- We first run Algorithm 2 to set the active cores and their frequencies
- Then we run the allocation algorithm on the single-threaded versions of the tasks.

The selected allocation algorithms are *Best Fit* (BF), *Worst Fit* (WF), *First Fit* (FF) and *Next Fit* (NF). Our algorithm is denoted as *Cut-Points* (CP). Notice that, if we consider tasks with only one single-thread cut-point, CP is equivalent to FF. Otherwise the main difference in CP is the decomposition. Notice also that decomposition make it easier to allocate threads, but increases the total utilisation factor because of the added synchronization overhead.

Figure 5 describes the average cumulative strength $\mathcal{S}$ of all algorithms by varying the utilisation factor. Each point represent the average value of $\mathcal{S}$ on 500 run of randomly generated task sets.

Algorithm 6 TaskSetGeneration

Input: m,n, K: Int,Cost:Float, T[n]:Int

Output:: taskSet

taskSet = $\emptyset$

U:Array[Float]=generateUtilizationRates()

for $(i = 0\ to\ n-1)$ **do**

$\mathcal{C} = T(i) * U(i)$;

$prob = generateBetween(7, 15)$

$ct = C * (1 - prob)$;

$mt = C * prob$;

if $(U(i) \leq 1)$ **then**

$D = generateBetween(C, T)$

else

$D = generateBetween(0.75, 1) * T[i]$

end if

k=generateBetween(1,K);

for $(k = 0\ to\ K-1)$ **do**

Z=generateBetween(2,m);

UZ = generateUtilizations(Z,1+Cost * Z);

for $(z = 0\ to\ Z-1)$ **do**

$mt_z = mt \times \mathrm{UZ}_z$;

$ct_z = ct \times \mathrm{UZ}_z$;

add Thread to CP

end for

add CP to Γ_i

end for

add Task to tasklist

end for

return toretTS;

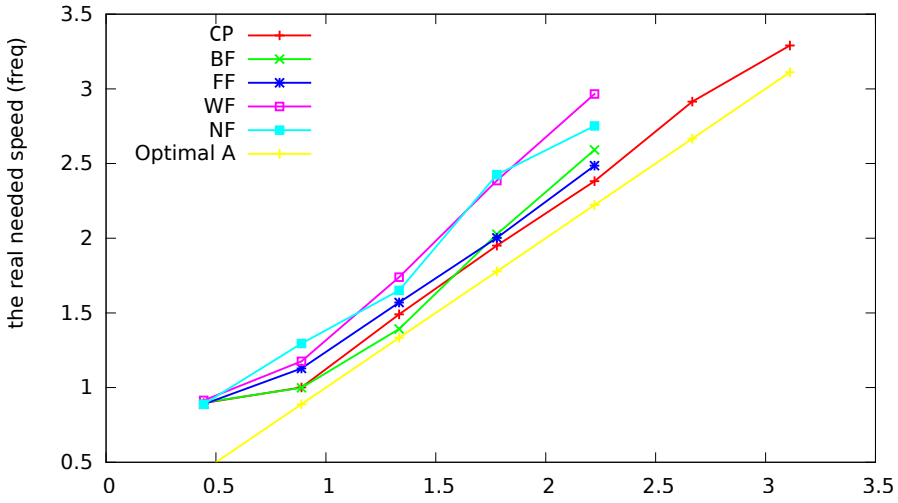

Figure 5: Cumulative strength by different utilisation factors.

We notice that our heuristic dominate globally the BF, WF, FF, NF heuristics. However, we can notice that when the workload is small, BF and FF are very close to CP, because the non-parallel heuristics (BF, WF, FF, NF) allocates the tasks entirely in one core (no-parallelism) and so they do not run the overhead brought by parallelizing the task. However, CP is stills better even when the computation load is not very high. When the load increases, BF, WF, FF, and NF are too much limited compared to our approach. In particular, our approach is close to the theoretical limit of full utilisation per each core, so the cumulative strength is almost equal to the task set utilisation (the yellow line).

Figure 6 describes the average cumulative strength $\mathcal{S}$ by varying the percentage of memory access ($\mathsf{mt}/(\mathsf{ct}+\mathsf{mt})$) for the same utilisation ($U = 2$). Once again, every point represents the average value of $\mathcal{S}$ obtained over 500 runs on randomly generated task sets. We notice that a small variation in memory accesses produces a considerable impact on the cumulative strength. However, the strength grows slowly after some percentage because the scheduling heuris-

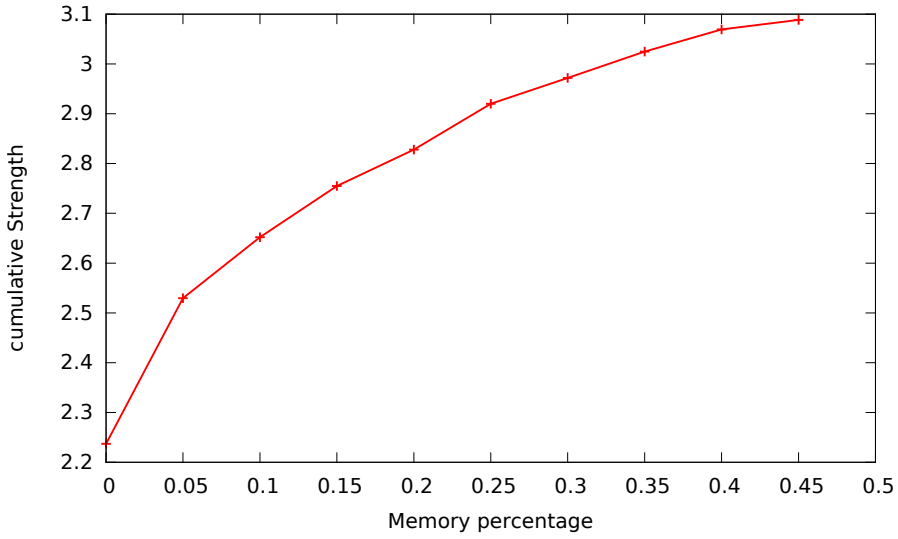

Figure 6: Architecture selected cumulated strength by different Memory rates.

tic behaves in the same manner on all cores and the core speed has less influence on the decomposition and allocation.

Figure 7 describes the average power consumption by varying the utilization rate. We plot only BF, FF and CP because the WF and NF show very bad results. Even if the utilisation produced by CP is higher than the others (due to the overhead of decomposition), it still consumes less energy for all scenarios especially when the load is high. Whereas BF and FF raise the cores frequency to high values, our algorithm keeps the frequency of each core lower. Even when the computation load is low, our approach saves in average 46% , 36% less than BF, FF.

Figure 8 describes the the utilization factor per each core while varying the utilization of the task set. We can notice that when the load is greater then the minimal speed (0.44), the utilization-rate per core gets closer to 1 (it reaches 97%), which means that the optimal solution is not so far.

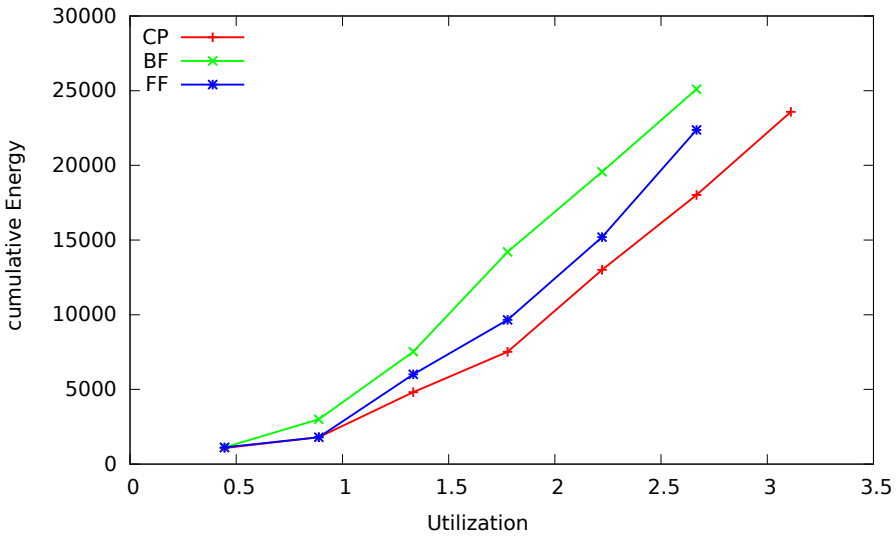

Figure 7: Energy Consumption.

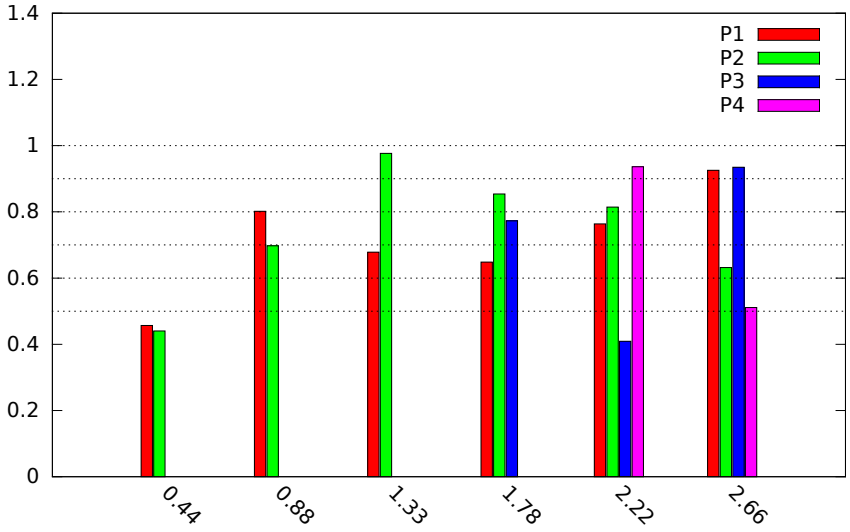

Figure 8: Per core utilization factor.

7. Conclusion

In this paper we presented a method for minimizing the energy consumption for a realistic task model on multicore uniform architectures. We propose a general model of a multicore, where cores are partitioned in groups, and cores belonging to the same group must share the same frequency. We also allow some processor to go in deep low power state. We also proposed a model for moldable tasks that easily allow to specify parallel decomposition (in the style of OpenMP) taking into account the synchronization overhead and a sound energy consumption model.

Then we proposed a heuristic algorithm for energy-aware EDF-partitioned scheduling, which selects the set of active cores and their frequency, and allocates tasks so that the resulting schedule is feasible. We have performed experiments that show the effectiveness of our algorithm.

In the future, we plan to implement our methodology on the ARM big.LITTLE architecture. Furthermore, we are working on an OpenMP-like API based on pthread for programming moldable tasks.

References

[1] F. Bonomi, R. Milito, J. Zhu, S. Addepalli, Fog computing and its role in the internet of things, in: Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing, MCC '12, ACM, New York, NY, USA, 2012, pp. 13–16. doi:10.1145/2342509.2342513.
URL http://doi.acm.org/10.1145/2342509.2342513

[2] M. Wolf, Computers as components: principles of embedded computing system design, Elsevier, 2012.

[3] R. Chandra, Parallel programming in OpenMP, Morgan Kaufmann, 2001.

[4] J. Goossens, V. Berten, Gang ftp scheduling of periodic and parallel rigid real-time tasks, arXiv preprint arXiv:1006.2617.

[5] M. Drozdowski, Scheduling Parallel Tasks Algorithms and Complexity, chapter 25. Handbook of SCHEDULING Algorithms, Models and Performance Analysis, CHAPMAN and HALL/CRC, 2004.

[6] P. D. G. Mounie, D. T. M. Drozdowski, Scheduling Parallel Tasks Approximation Algorithms, chapter 26. Handbook of SCHEDULING Algorithms, Models and Performance Analysis, CHAPMAN and HALL/CRC, 2004.

[7] E. Bini, G. Buttazzo, G. Lipari, Speed modulation in energy-aware real-time systems, in: Real-Time Systems, 2005.(ECRTS 2005). Proceedings. 17th Euromicro Conference on, IEEE, 2005, pp. 3–10.

[8] R. Jejurikar, C. Pereira, R. Gupta, Leakage aware dynamic voltage scaling for real-time embedded systems, in: Proceedings of the 41st annual Design Automation Conference, ACM, 2004, pp. 275–280.

[9] P. Pillai, K. G. Shin, Real-time dynamic voltage scaling for low-power embedded operating systems, in: ACM SIGOPS Operating Systems Review, Vol. 35, ACM, 2001, pp. 89–102.

[10] J.-J. Chen, C.-F. Kuo, Energy-efficient scheduling for real-time systems on dynamic voltage scaling (dvs) platforms., in: RTCSA, 2007, pp. 28–38.

[11] D. Zhu, R. Melhem, D. Mossé, The effects of energy management on reliability in real-time embedded systems, in: Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on, IEEE, 2004, pp. 35–40.

[12] K. Lakshmanan, S. Kato, R. Rajkumar, Scheduling parallel real-time tasks on multi-core processors, IEEE, 2010, pp. 259–268. doi:10.1109/RTSS.2010.42.
URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5702236

[13] P. Courbin, I. Lupu, J. Goossens, Scheduling of hard real-time multi-phase multi-thread (MPMT) periodic tasks, Real-Time Systems 49 (2) (2013)

239–266. doi:10.1007/s11241-012-9173-x.
URL http://link.springer.com/10.1007/s11241-012-9173-x

[14] S. Kato, Y. Ishikawa, Gang EDF scheduling of parallel task systems, IEEE, 2009, pp. 459–468. doi:10.1109/RTSS.2009.42.
URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5368128

[15] G. Manimaran, C. S. R. Murthy, K. Ramamritham, A new approach for scheduling of parallelizable tasks in real-time multiprocessor systems, Real-Time Systems 15 (1) (1998) 39–60.

[16] S. Han, M. Park, Predictability of least laxity first scheduling algorithm on multiprocessor real-time systems, in: Emerging Directions in Embedded and Ubiquitous Computing, Springer, 2006, pp. 755–764.

[17] S. Collette, L. Cucu, J. Goossens, Integrating job parallelism in real-time scheduling theory, Information Processing Letters 106 (5) (2008) 180–187.

[18] A. Saifullah, J. Li, K. Agrawal, C. Lu, C. Gill, Multi-core real-time scheduling for generalized parallel task models, Real-Time Systems 49 (4) (2013) 404–435. doi:10.1007/s11241-012-9166-9.
URL http://link.springer.com/10.1007/s11241-012-9166-9

[19] N. Fisher, J. Goossens, P. M. Hettiarachchi, A. Paolillo, Energy minimization for parallel real-time systems with malleable jobs and homogeneous frequencies, arXiv preprint arXiv:1302.1747.
URL http://arxiv.org/abs/1302.1747

[20] K. Seth, A. Anantaraman, F. Mueller, E. Rotenberg, Fast: Frequency-aware static timing analysis, ACM Transactions on Embedded Computing Systems (TECS) 5 (1) (2006) 200–224.

[21] A. Peter Greenhalgh, White paper: big.little processing with arm cortex a15 - cortex a7, Tech. rep., ARM, East Lansing, Michigan (September 2011).

[22] J. Mei, K. Li, J. Hu, S. Yin, E. H.-M. Sha, Energy-aware preemptive scheduling algorithm for sporadic tasks on DVS platform, Microprocessors and Microsystems 37 (1) (2013) 99–112. doi:10.1016/j.micpro.2012.11.002.
URL http://linkinghub.elsevier.com/retrieve/pii/S0141933112001895

[23] S. K. Baruah, L. E. Rosier, R. R. Howell, Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor, Real-Time Systems 2 (4) (1990) 301–324.

[24] I. Cplex, 11.0 users manual, ILOG SA, Gentilly, France.

[25] R. H. Byrd, J. Nocedal, R. A. Waltz, Knitro: An integrated package for nonlinear optimization, in: Large-scale nonlinear optimization, Springer, 2006, pp. 35–59.

[26] R. Dais, A. Burns, Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems, IEEE, 2009, pp. 398–409. doi:10.1109/RTSS.2009.31.
URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5368139

[27] J. Goossens, C. Macq, Limitation of the hyper-period in real-time periodic task set generation, in: In Proceedings of the RTS Embedded System (RTS01, Citeseer, 2001.

[28] H.-S. Yun, J. Kim, On energy-optimal voltage scheduling for fixed-priority hard real-time systems, ACM Transactions on Embedded Computing Systems (TECS) 2 (3) (2003) 393–430.

*Author Biography & Photograph

Energy-aware Real-Time Task Decomposition for partionned-EDF Scheduling on Multi-core Uniform Architectures

Houssam Eddine ZAHAF[a,b], Richard OLEJNIK[a], Abou ElHassen BENYAMINA[b], Giuseppe LIPARI[a]

[a] *Lille University, France*
[b] *Oran 1 University, Algeria*

Houssam Eddine ZAHAF received his Licence degree in computer science from Oran 1 University,Oran, Algeria, in 2011. He received his master degree in 2013 in embedded systems and is currently a Ph.D. Candidate in the Department of computer science at Lille, France and Oran 1, Algeria Universities. His research interests are in the areas of real-time systems, operating system, and scheduling with emphasis on mathematical modeling, performance analysis and energy consumption.

Benyamina Abbou el hassen graduated from Department of Computer Science, Faculty of Sciences, University of Oran, Algeria, where he received PhD degree in computer science in 2008. He is currently professor of computer science in the Univ. Es-senia ORAN, Sciences et Technologies, Algeria. He is head of the LAPECI team. His research works include parallel processing, optimization, design space exploration and Model Driven Engineering with the special focus on real-time and embedded systems.

OLEJNIK Richard . is a researcher in the Computer Science Laboratory of Lille Cristal, got his PhD on 1984 on Computer Graphics and have been working since as Research Engineer for the French National Centre of Researches (CNRS) at University of Lille I. He is interesseted in Computer Architecture, Distributed Systems, Grid and High performance Computing and embedded system. He manages an international projects on middleware and tools for Grid and Datamining computing (SOAJA project : Service Oriented Adaptative Java Applications middleware) and (PICS 4334 an international project beetwen the French National Research Center (CNRS) and the Polish Academy of Sciences, about "New approaches for load balancing and scheduling in computational grids and embedded systems").

LIPARI Giuseppe is Full Professor of Computer Science at University of Lille. He is part of the Embedded Real-Time Adaptative system Design and Execution (meraude) team of the Centre de Recherche en Informatique, Signal et Automatique (CRIStAL) de Lille. He is Senior IEEE member, and associate editor of the Journal of System Architecture and of the Real-Time Systems Journal. His research interests are in real-time systems, real-time operating systems, scheduling algorithms, embedded systems, wireless sensor networks.