Energy Efficient Online Scheduling of Real Time Tasks on Large Multi-threaded Multiprocessor Systems

M. Ghose*, A. Sahu, S. Karmakar

Department of Computer Science and Engineering
IIT Guwahati
India

Abstract

Today's large compute systems are empowered with high processing capabilities and they are often used to run real time applications. But these systems consume significantly large amount of energy while executing these applications. Thus reducing energy consumption for such systems becomes as important as meeting the deadline constraints of the applications they execute. Scheduling using DVFS is proved to be a promising technique for reducing energy consumption of the systems executing real time applications. But this technique is not as efficient for large systems as it is for small systems and hence there is a need to design scheduling techniques for large systems considering power construct at a coarser granularity. In this paper, we have exploited the actual power consumption pattern of a few recent commercial multi-threaded processors and derived a simple power model which considers the power consumption at a coarser granularity. We have then proposed an online energy efficient task scheduling policy namely, *smart allocation policy* for scheduling aperiodic independent real time tasks onto such large systems having multi-threaded feature in the processors. We have further added three variations of our proposed policy to efficiently address different situations which can occur at execution time and to further reduce energy for some particular kinds of applications. We have analyzed the instantaneous power consumption and the overall energy consumption of four proposed task allocation policies along with other five baseline policies for a wide variety of synthetic data sets and real trace data. As the computation time of tasks plays a major role in scheduling and effects the overall performance of the system, we have considered six different computation time models in our work. Also different applications exhibit different kinds of deadline behavior and thus we performed our experiments with five different kinds of deadline schemes. Experimental results show that our proposed policies achieve average energy reduction of 45% (maximum up to 92%) for synthetic data set and 30% (maximum up to 47%) for real data sets as compared to baseline policies. All the proposed policies ensure that no task misses its deadline.

Keywords:
real time tasks, online scheduling, energy efficient, large systems

1. Introduction

Now a days, the number of processors and number of threads per processor have increased to a significant number in compute systems. Thus the processing capability of these systems are sufficient enough to handle most of the recent applications. But the major concern in these computing systems is the growing energy consumption; which has sought the attention of the research community to a great extent. These compute systems include battery operated mobile devices, desktops and servers. Though the processing capabilities of most of the data centers are sufficient enough, they remain under utilized most of the times. This is same with the mobile devices and desktop computers. These underutilized systems unnecessarily consume a significant amount of power (or energy). When power consumption in the servers and workstations increases, heat dissipation also increases. Thus reducing the power consumption in servers and workstations not only reduces the operating electricity cost but also lowers the cooling cost. Reducing power consumption increases the battery life for hand-held devices and there is a constantly increasing market demand for the equipments consuming lesser amount of power.

In many compute systems, we require to deal with real time tasks where executing the tasks before their deadline is essential. Scheduling real time tasks in multiprocessor domain is a promising research area in recent time. But the traditional multiprocessor real-time scheduling algorithms aim to (a) improve utilization bound, (b) reduce approximation ratio, (c) reduce resource augmentation and (d) improve some empirical factors, like total schedule length, total laxity, number of deadline misses, etc [1, 2]. But the current research trend is to associate the power consumption of the processors with scheduling and the goal is to design scheduling algorithms which reduce the power (or energy) consumption of the

*Corresponding author
Email addresses: g.manojit@iitg.ernet.in (M. Ghose), asahu@iitg.ernet.in (A. Sahu), sushantak@iitg.ernet.in (S. Karmakar)

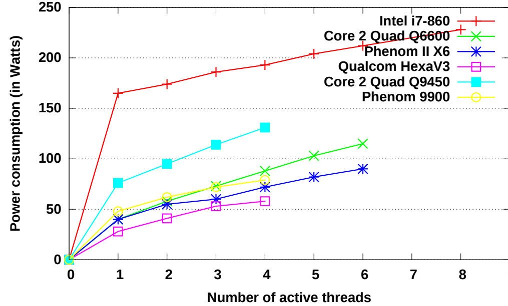


Figure 1: **Power consumption plot of various commercial processors with different number of active hardware threads [12, 13]**

processing elements. These algorithms are commonly termed as power aware scheduling or energy efficient scheduling [3, 4, 5].

Though there are many power consuming components present in a computing system, a major portion of the power is consumed by the processor itself. Thus the research mainly focuses on the power consumption of the processor. Again there are three components in processor power consumption: dynamic, static and short-circuit, and research separately targets individual power consumption components to reduce overall power of the processor [6]. But in this paper, we have derived a power model based on the actual power consumption behavior of commercial processors. Thus it automatically caters all the components of the power consumption together.

Traditional power aware or energy efficient scheduling techniques use the dynamic voltage and frequency scaling (DVFS) to design their algorithms [7]. They consider the power model as: $P \propto f^3$ where P is the power consumption of a processor and f is the operating frequency of that processor. DVFS focuses on power consumption behavior of the processor at a finer granularity. On the other hand, in large scale computing systems (both servers and clouds), we need to consider the power consumption model at a coarser granularity. In the era of dark silicon, power consumption behavior is handled at the higher granularity by completely switching off or on a computing region (or processor (or a set of processors) based on the requirements to minimize the power consumption [8, 9]. Similar concepts can be seen profoundly in cloud systems which manage the virtual machines [10, 11]. In this case, the common idea is to run as less physical nodes as possible.

Figure 1 shows a plot of the power consumption of various commercial processors with different number of active hardware threads. We are motivated by the power consumption behavior of these recent multi-threaded processors. We can easily observe a relationship between the power consumption of a processor and the number of active hardware threads of that processor. Processor consumes a significant amount of power when it gets started and runs one thread; after that the power value increases almost linearly with the increase in number of active threads till the thread number reaches processor's maximum capacity. The power consumption of Qualcom Hexa V3 with one active thread is 40 watts. When it runs two threads, the power consumption value becomes 55 watts. This 15 watts ($= 55-40$) is the extra power consumption for the second thread. Similarly, subsequent threads consume additional power values as 5 watts ($= 60 - 55$), 12 watts ($= 72 - 60$), 10 watts ($= 82 - 72$) and 8 watts ($= 90 - 82$) respectively. Observing this power consumption pattern of the recent processors, we have derived a simple power model for a multi-threaded single processor as described in Section 3.

Another motivation to use high level power consumption construct is that in recent days commercial processors have switch-off power state (example SOix: System S0 power state of Intel 6th generation U-series processors) and these kind of lower energy consumption processors are getting used in large data centers. In Intel 6th generation U-series processors, the S0ix state (System S0 power management state) enable the CPU of a connected standby system to enter the deepest state by turning the supply off [14, 15]. To the best of our knowledge, no scheduling algorithm on real time tasks were proposed considering the thread based power consumption pattern of commercial processors as explained in Figure 1. We believe that our derived power consumption model is well suited for Large Multi-threaded Multiprocessor Systems (LMTMPS).

Our contributions in this paper can be summarized as follows.

- We have exploited the power consumption pattern of the commercial processors and derived a simple power consumption model based on this pattern, and used this model in our work.

- We have designed four online energy efficient scheduling policies namely (a) *smart allocation policy*, (b) *smart - Early Dispatch allocation policy*, (c) *smart - Reserve allocation policy* and (d) *smart - Handling Immediate Urgency allocation policy* which use the derived power model to reduce the overall energy consumption while meeting the deadline constraints of all the tasks.

- As the computation time and deadline of tasks play a significant role in scheduling, we have considered six different computation time schemes and five different deadline schemes under the task model.

- We have compared and analyzed the performance of these policies with three other standard policies, the standard EDF policy and state of art utilization based work consolidation (UBWC) policy, and found that our proposed policies consume significantly less energy consumption compared to all baseline policies both for synthetic task sets and real workload traces.

Rest of the paper is organized as follows: Section 2 provides a brief description of previous work in the area of power aware or energy efficient scheduling techniques. Section 3 describes the problem formulation, underlying model of

compute system, task system model and used data sets. Section 4 and Section 5 explains different existing and proposed task allocation policies in details. Section 6 describes about the experimental setups, experiments and analysis of result. Finally, we concluded the paper and discussed about future directions in Section 7.

2. Previous Work

In this section, we present a brief overview of the research works done in the context of energy efficient or power-aware scheduling. The work can be classified into mainly two categories (a) fine-grained approach, (b) coarse-grained approach. In fine-grained approach, research mainly targets to reduce the power consumption of the cores by exploiting the DVFS techniques. This approach is driven by the fact that the actual computation time of a task is often less than the worst case computation and the algorithms make use of this slack time in different ways. On the other hand, coarse-grained approach works at the processor level for the small systems and at the host level or server level for the large systems (host or server have many processors). Coarse-grained approach mainly focuses on reducing the number of active hosts and putting the passive hosts in low power state or in sleep mode.

Weiser et al. [16] was pioneer to start the research in this direction by associating power consumption with scheduling and used DVFS technique to study the power consumption of some scheduling techniques. They took advantage of CPU idle time and reduced the operating frequency of CPU so that the tasks were finished without violating any deadline. Aydin et al. [17, 18] ported this idea for real time systems where they initially designed a static algorithm for computing the optimal frequency for a periodic task set assuming their worst case behavior. After that they devised an online speed adjustment algorithm for dynamically claiming the energy not used by the tasks finishing earlier than the worst case time and achieved up to 50% power saving as compared to the worst case scenario. Zhu et al. [19] has extended the technique of [17] for the multiprocessor environment. In their approach, they utilize the slack created by a task to reduce the energy consumption and the slack can be (a) utilized by another task in the same processor, (b) shared between different processors.

Isci et al. [20] has proposed a global power management policy for chip multiprocessors where every processor can operate in three different modes: turbo (no reduction), eff1 (5% voltage and frequency reduction), eff2 (15% voltage and frequency reduction). They studied the relationship between the operating modes of the processors and the corresponding performances. Lee and Zomaya [21, 22] have proposed makespan-conservative energy reduction along with simple energy conscious scheduling to find a trade-off between the makespan time and energy consumption, where they reduced both makespan time and energy consumption of precedence constraints graph on heterogeneous multiprocessor supporting DVFS.

Recently, Li and Wu [23, 24, 25] has considered execution of various task models by further exploiting the DVFS technique for both homogeneous and heterogeneous processor environments. All of these studies consider the DVFS properties for the processors and do not consider the multi-threaded feature of the processors. In contrast, we consider a set of non-DVFS processors having multiple hardware threads and attempt to minimize the energy consumption.

Chase et al. [26] proposed a coarse-grained power management techniques for Internet server clusters where an energy conscious switch is placed on top of the servers that maintains a list of active servers. Depending on the traffic behavior and a predetermined threshold value of utilization of the servers, the switch dynamically adjusts the list by keeping them either in active mode or low power mode. In [27] they have extended the idea to develop an economic model based architecture, called *Muse* to find the trade-off between the service quality and the cost (i.e. energy). Hotta et al. [28] designed a scheme for the high performance computing (HPC) clusters where the program execution is splitted into multiple regions and the optimal frequency is chosen for individual region. Srikantaiah et al. [29] studied the relationship between the energy consumption and the performance of the system which was determined by the CPU (or processor) utilization and disk utilization. The problem was viewed as a bin packing problem where applications are mapped into servers optimally. Kokkinos et al. [30] has studied data consolidation problem where a task requires multiple pieces of data scattered in different nodes of a grid network. Choi et al. [31] studied the characteristics of the applications running in data centers in order to impose appropriate bounds on power consumption. Verma et al. [32] studied the power consumption pattern of various HPC applications and the typical workload behavior of a virtualized server and in [33] they developed a framework called pMapper where the applications are placed onto different servers for execution based on the utilization values of the servers.

Our work is similar to the above mentioned research but it differs from others in the sense that we have considered the execution of real time tasks in large systems where we try to minimize the overall energy consumption with a guarantee of real-time constraints in a multi threaded multiprocessor environment.

3. Problem Formulation: System Model, Task Model, Data Sets

3.1. Problem Formulation

We wish to design scheduling policies for online aperiodic independent tasks onto large multi-threaded multiprocessor systems such that no task misses its deadline and the total energy consumption of the system is minimized. In this work, we assume that the compute capability of the system is high i.e. the number of processors is taken as sufficiently large. Power consumption in the multi-threaded multiprocessor systems under the considered model is not proportional to the utilization of the system (or number of active hardware threads on the system) as there is static component in the power model and the total power value also depends on the number of active processors. In this work, we have used this power consumption behavior to reduce the total energy consumption of the system.

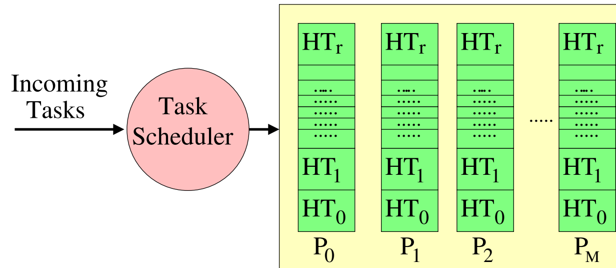


Figure 2: **Scheduling of online aperiodic tasks on large multi-threaded multiprocessor system**

3.2. System Model

Figure 2 shows the block diagram of our considered system which schedules and executes a set of online aperiodic real time tasks on large multi-threaded multiprocessor system. Tasks arrive to the system dynamically (online aperiodic tasks) and the scheduler schedules the tasks on to the processing system. Our considered processing system consists of M number of multi-threaded processors (P_0, P_1, ..., P_M) where M is sufficiently large. It is justified to assume a system with such large number of processors, as the modern days computing systems like cloud system, promise virtually infinite resources to all the applications. All the processors are homogeneous, non-DVFS capable and each processor is multi-threaded having r hardware threads. In this paper, hardware thread of a processor is also termed as virtual processor.

We have also assumed that one task gets allocated to exactly one hardware thread of a processor and one thread can execute only one task at a time. Once all the hardware threads of a processor is filled with tasks, a new processor is switched on. If a hardware thread of any processor becomes free in between, a new task is allocated to that hardware thread or an existing task may be migrated to this processor based on the situation.

As mentioned in Section 1 of the paper, we have exploited the power consumption behavior of a few commercial multi-threaded processors and derived a simple power model where each processor consumes a significant amount of power as it gets started. This significant amount is termed as the *base power consumption* of the processor. Each active thread contributes some amount to power called the *thread power consumption* whose value is significantly lesser than the base power consumption of the processor. The processor power consumption of such large multi-threaded multiprocessor systems (LMTMPS) can be modeled using the derived power model which is described using equation 1 and 2.

For a single multi-threaded processor, power consumption can be modeled as:

$$P_s = C + i\delta \tag{1}$$

where, P_s = power consumption of a single processor, C = base power consumption of the processor, i = number of active hardware threads and δ = thread power consumption (or power consumption per active thread) . In general, the value of C is greater than δ. Typical values of C is 5 to 10 times of δ [12, 13].

Similarly, the power consumption for the whole LMTMPS can be modeled as:

$$P_{LMTMPS} = L.(C + r\delta) + (C + i\delta) \tag{2}$$

where, P_{LMTMPS} = total power consumption of the large multi-threaded multiprocessor system, L = number of switched-on processors which are utilized fully and r is the maximum number of hardware threads a processor can run.

The first part of the equation 2 represents the total power consumption of the fully utilized processors and the second part $C + i\delta$ indicates the total power consumption of the partially filled processor. If there is no partially filled processor, then the second part of the equation evaluates to zero. Fully utilized (filled) processor means all the hardware threads of the processor are active and executing some tasks, otherwise the state of the processor is partially filled or partially utilized. Here we assumed that our task scheduler regularly (or frequently) runs through the system and consolidates the running tasks into fewer number of processors. In this process, neighbouring processors are chosen so as to reduce the power wastage due to its distribution. The tasks are consolidated and migrated to other processors in such a way that (a) only one processor or no processor will be partially filled, and (b) other processors will be either completely filled or in switched off mode. Figure 3 shows the power consumption behavior of LMTMPS. We can see that there is a sharp jump in power consumption value when the number of virtual processors (or active hardware threads) is 1, 9 and 17. In most of the earlier proposed scheduling policies, researchers considered power model where power consumption is proportional to utilization. But in practice the power consumption is not exactly proportional to the utilization because the static component present in the total power model contributes a significant amount. Gao et al. [34] considered similar types of power model for the virtualized data centers where they considered power consumption of a server as a function of its utilization and here we considered power consumption of a processor as a function of number of active hardware threads but not proportional to the number of active hardware threads.

3.3. Task Model

In this paper, we have considered scheduling a set of online independent aperiodic real time tasks, $\mathcal{T} = \{T_1, T_2, \ldots T_N\}$ onto a large multi-threaded multiprocessor system. Each task T_i is characterized by its arrival time (a_i), computation time

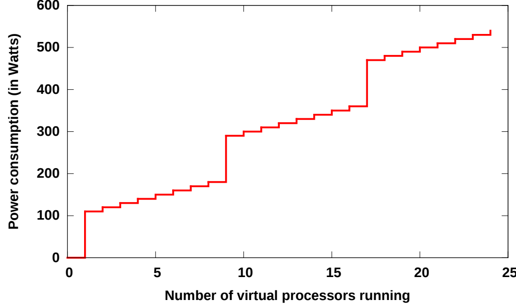


Figure 3: **Power consumption of LMTMPS under the proposed model $C = 100, \delta = 10$ and $r = 8$**

(c_i) and deadline (d_i) where $d_i \geq (a_i + c_i)$. We assume that all the tasks are sequential and it can be run by only one hardware thread (virtual processor) at any time instant.

3.4. Data Sets: Synthetic Data and Real Workload Traces

In this paper, we have considered scheduling of both synthetic real time tasks and real trace data which are described in following subsections.

3.4.1. Synthetic Tasks

Here we assume that a set of independent aperiodic real time tasks are arriving to the system in such a way that the inter-arrival time between any two consecutive tasks follow discretized Gaussian distribution. As the execution (or compute) time of tasks is a key factor in scheduling and it has a significant impact on the overall performance of the system [35, 36], we have considered a wide variety of task computation time with different distributions in order to establish the effectiveness of our work. Here we have considered four different distributions: **Random, Gaussian, Poisson** and **Gamma**. In random distribution, the execution time or computation time c_i of a task T_i is uniformly random distributed between 1 to C_{max}, where C_{max} is a user defined value. We have further considered computation time as a function of time or task sequence number: $INC(i)$ and $DEC(i)$. In $INC(i)$, computation time of tasks increases with task number. $INC(i)$ is taken as $k.i$, $i \in \{1, 2, \ldots, N\}$ and $c_i \geq c_{i-1}$. Similarly, in $DEC(i)$, computation time of tasks decreases with task number. $DEC(i)$ is taken as $k.(N + 1 - i)$, $i \in \{1, 2, \ldots, N\}$ and $c_i \leq c_{i-1}$. In addition to these variations of computation time, we have considered five different variations in deadline of tasks. An application can be considered as a collection of such tasks and different kinds of application exhibit different deadline behaviors with time. For a task T_i (a_i, c_i, d_i), $d_i - (a_i + c_i)$ is called the slack time (SLK) of the task and it can be a constant, a function of time or sequence, or something else which depends on the deadline schemes. Different deadline schemes considered in our work is explained below.

1. **Scheme 1: Randomly distributed deadline**
   In this scheme, the slack time for all the task is randomly distributed. Deadline of a task T_i arriving at time a_i is $d_i = a_i + c_i + z$; where the slack time z is a random number, varies in the range 0 to Z_{max}. These are the tasks having similar relative deadlines. Relative deadline of a task is the absolute deadline of the task minus its arrival time.
2. **Scheme 2: Gaussianly distributed deadline**
   In this scheme, deadline of a task $d_i = a_i + c_i + g(\mu, \sigma)$; where $g(\mu, \sigma)$ is slack function which returns an integer, and this function follows discretized Gaussian distribution with mean μ and variance σ.
3. **Scheme 3: Increasing deadline**
   In this scheme, relative deadline of tasks increase with time (or task sequence). It can be represented as $d_i = a_i + c_i + INC(i)$; where i is task sequence, $INC(i)$ is increasing function with i, $i \in \{1, 2, \ldots, N\}$, and N is total number of tasks. These are the tasks where deadline is very tight initially and it is relaxed as the time progresses (and task sequence ultimately increases). This models the interest of people for a new product (or movie) release or of tasks of similar kind. Initially, people are keen to buy a new product or watch a new movie and as time progresses, interest comes down.
4. **Scheme 4: Decreasing deadline**
   In some cases, we see that the relative deadline (or slack) of tasks decreases with task sequence (or time). This scheme represents this kind of scenario. Mathematically, it can be represented as $d_i = a_i + c_i + DEC(i)$; where i is task sequence, $DEC(i)$ is a decreasing function with i, $i \in \{1, 2, \ldots, N\}$, and N is total number of tasks. This is basically an opposite case as that of previous scheme. Here the deadline is relaxed initially and as the time progresses, deadline becomes tighter. This scenario depicts the form submission on or before a specified deadline.
5. **Scheme 5: Common deadline**
   This is the scheme where the absolute deadlines of the tasks do not change with task sequence. It can be written as $d_i = D$; where D is the common deadline for all the tasks satisfying the condition $D \geq max\{a_i + e_i\}$. This is termed as common due date problems in literature [37].

3.4.2. Real Workload Traces

In addition to synthetic real time tasks with many variations (as described in previous subsection), we have also considered four different real-life workload traces (or logs), namely, CERIT-SC, MetaCentrum-1, Zewura and MetaCentrum-2 workload traces [38] in our work. These workload traces (or data sets) are generated from TORQUE and PBSpro traces in different times which contain the job descriptions of different clusters in Standard Workload Format (SWF) [39]. CERIT-SC workload trace consists of 17,900 jobs collected during the first 3 months of the year 2013. MetaCentrum-1 workload trace contains job descriptions of 495,299 jobs which are collected during the months of January to June of the year 2013. Zewura workload trace contains 17,256 jobs of 80 CPU clusters which are collected during the first five months of the year 2012. MetaCentrum-2 workload trace consists of 103,656 jobs descriptions of 14 clusters containing 806 CPUs which are collected during the first five months of the year 2009. The logs also contain several other useful information such as node descriptions, queue descriptions, machine descriptions, etc. These logs and data sets are provided by the Czech National Grid Infrastructure MetaCentrum [38].

4. Standard Task Allocation Policies

In this section, we present three standard task allocation policies, namely, utilization based allocation policy, front work consolidation and rear work consolidation. We also state the standard earliest deadline first policy and describe the state of art task allocation policy, namely, utilization based work consolidation policy in details.

4.1. Utilization Based Allocation Policy (UBA)

In this policy, tasks are allocated to the processors based on their processor utilization value. Processor utilization of a task can be defined as the amount of processor share required to finish its execution *just-in-time* (i.e. on its deadline). Processor utilization u_i of a task T_i can be defined by the equation 3.

$$u_i = \frac{c_i}{d_i - a_i} \tag{3}$$

where, c_i is the computation time, d_i is deadline and a_i is the arrival time of the task T_i.

Whenever a tasks arrives at the system, it is allocated a portion of the processor's share for its execution. This policy allows unrestricted number of migrations and preemptions of the tasks. Virtually, a task T_i gets executed in the system starting from its arrival time a_i till its deadline d_i but with minimum processor share u_i. The start time (s_i) and finish time (f_i) of a task T_i is given by equation 4.

$$s_i = a_i; \qquad f_i = d_i \tag{4}$$

At any instant of time t, number of virtual processors required to execute all the tasks and to meet their deadlines can be defined as the summation of utilization values of all the tasks present in the system. It can be written as

$$PN_t = \left\lceil \sum_{i=0}^{N_{active_tasks}} u_i \right\rceil \tag{5}$$

where, PN_t = number of virtual processors required and N_{active_tasks} = Total number of tasks currently running in the system (or the number of tasks for which $a_i \leq t \leq d_i$).

This policy provides the minimum processor share (i.e. utilization) required for a task for all the time instants starting from its arrival time till its deadline. Thus the execution of a task T_i spreads for the whole duration with minimum processor requirement u_i at every time instant between a_i and d_i. It aims to reduce the instantaneous power consumption of the processor by allocating the minimum amount of processor share. This policy is mainly of theoretical interest and the assumption made in Section 3 that one thread can execute exactly one task at a time does not hold here.

4.2. Front Work Consolidation (FWC)

In this policy the execution of tasks are consolidated towards beginning of the time axis. As soon as a task arrives, it is allocated to a virtual processor (a thread of a physical processor) and it starts its execution immediately. Here, start time (s_i) and finish time (f_i) of a task T_i is given by equation 6. As all the tasks start their execution immediately after they enter the system, the policy can also be termed as *Immediate Allocation Policy* or as soon as possible (ASAP) policy. As the system has sufficient number of processors, immediate allocation is always feasible. This policy turns out to be a *non-preemptive* one and there is no *migration* of tasks.

$$s_i = a_i; \qquad f_i = a_i + c_i \tag{6}$$

To describe the front work consolidation (FWC) policy, let us consider an example with three tasks $T_1(a_1 = 0, c_1 = 4, d_1 = 8)$, $T_2(1, 2, 4)$ and $T_3(4, 6, 12)$ as shown in Figure 4. We can calculate utilization of task T_1 (u_1=0.5) between time 0 and 8, utilization of task T_2 (u_2=0.667) between time 1 and 4, and utilization of task T_3 (u_3=0.75) between time 4 and 12. If we schedule these tasks based on utilization then at least one task will be in execution between time 0 to 12 and it is shown in the middle part of Figure 4. In front consolidation, we try to consolidate (i.e. summing up) all the utilizations towards the beginning of time axis and thus for all the tasks, utilizations get consolidated towards the arrival time of tasks. As we assume all the tasks are sequential and they can be executed on one processor at a time, consolidated utilization

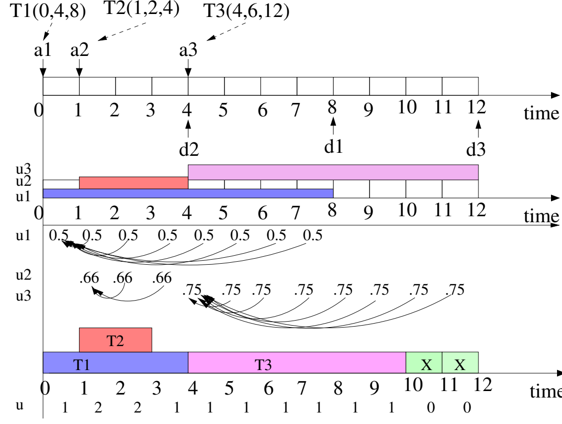


Figure 4: **Illustration of front work consolidation of real time tasks**

of one task T_i in any time slot can not exceed $u_i * \frac{d_i - a_i}{c_i}$. Consolidated utilization indicates the total amount of work to be done for a task with a particular utilization value. If the consolidated utilization exceeds this limit then it needs to be put into next time slot and the process is continued till there is no work left. In the above case, for task T_1, the total consolidated utilization becomes $u_1 * (d_i - a_i)$=0.5 $*$ 8=4, so it needs to be spread into four slots with consolidated utilizations $u_1 * \frac{d_1 - a_1}{c_i}$=0.5 $*$ (8 − 0)/4=1 in all the time slots from 1 to 4. Executing the example tasks set using this FWC policy, there will be no work in time slot 11 and 12. If there is no work, there is no power consumption (static power). This reduces overall energy of the system.

4.3. Rear Work Consolidation (RWC)

This policy is similar to the front work consolidation policy, but here the tasks are consolidated towards the end of the time axis. Every task execution gets consolidated towards the deadline of the task. This policy is also non-preemptive and does not require migration for system with sufficiently large number of single threaded processor.

As the tasks enter the system, they are accumulated and remain in the waiting queue as long as their deadlines permit them to wait. Execution of a task begins at the *urgent point* only. Urgent point of a task is the point in time axis where a task must start its execution in order to meet its deadline. This policy ensures that all the tasks can finish their execution just in time (i.e. on their deadlines). As the execution of all the tasks are delayed till their urgent points, the policy can also be termed as *Delayed Allocation Policy* and in this policy tasks are scheduled to the system as late as possible (ALAP) [40, 41]. Since we assume sufficiently large number of processors, delayed allocation is always feasible. And the start time (s_i) and finish time (f_i) of a task T_i is given by equation 7.

$$s_i = d_i - c_i; \qquad f_i = d_i \tag{7}$$

4.4. Utilization Based Work Consolidation (UBWC)

In this policy, the scheduler tries to schedule the currently arrived task $T_i(a_i, c_i, d_i)$ at any time instants (slots) in such a way that the task does not miss its deadline and there is minimum number of increase in the active processor count. For the task T_i, the UBWC schedules c_i units of unit time compute slot between the current time (a_i) and deadline of the task d_i, so that the increase in number of active processors count will be minimum in all the time slots between a_i and d_i [42, 29, 31, 30]. Processor count in a time slot increases when the total utilization of that time slot crosses a whole number. For r threaded processor, we can safely assume that the utilization of any time slot t increases by $\frac{1}{r}$ when a unit compute time of a task got scheduled at that time slot. In a system with virtually infinite single threaded processor, the UBWC will consume same amount of energy as UBA, FWC and RWC. But in systems with infinite multi-threaded processors, it reduces the energy consumption significantly because the task execution is serial and in every time instant the policy aims to keep the processor count minimum. When there is minimum processor count, the static component of the energy which is a significant portion also becomes minimum. This reduces the overall energy consumption.

Every incoming task to the system has arrival time a_i, deadline d_i and computation time c_i. When a task T_i arrives at system at a_i, there may be some tasks already present in the system and their remaining part of execution must have been scheduled between a_i and $\max\{d_j\}$, where d_j is the the deadline of the task T_j currently present in the system except T_i. For system with sufficiently large number of multi-threaded processors having r-hardware threads per processor, the UBWC scheduler inserts $c_i * \frac{1}{r}$ units of computation between a_i and d_i with a preference to early slots, so that it minimizes the number of active processors in all the time slots between a_i and d_j. This scheduling policy requires preemption and migration of the tasks; but the number of preemptions and number of migrations for a task T_i is bounded by $c_i - 1$ if we assume the time axis is discretized in unit time slot.

4.5. Earliest Deadline First Allocation Policy (EDF)

Earliest deadline first (EDF) is a well-known scheduling policy where tasks are considered based on their deadline values. Task with the earliest deadline value is executed first [2]. At any point of time t, we consider all the arrived tasks

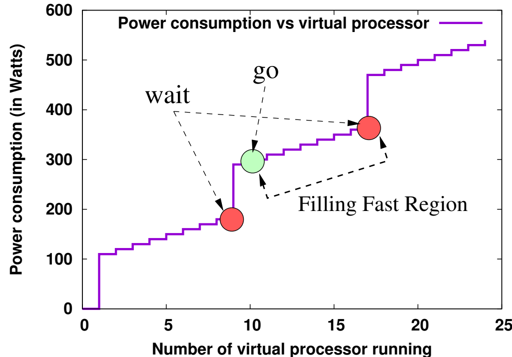


Figure 5: **Extra annotation to Figure 3 to describe the smart allocation policy ($C = 100, \delta = 10$ and $r = 8$)**

which have not started their execution. Out of these waiting tasks, task with the minimum deadline value is chosen for execution. We need to use minimum number of processors at any instant of time such that no task misses its deadline. The start time and finish time follow the inequalities 8 and 9.

$$a_i <= s_i <= (d_i - c_i) \tag{8}$$

$$(a_i + c_i) <= f_i <= d_i \tag{9}$$

5. Proposed Task Allocation Policies

We have proposed four energy efficient online task scheduling policies for executing a set of aperiodic independent real time tasks onto large multi-threaded multiprocessor system (LMTMPS), where instantaneous system power consumption is not proportional to instantaneous utilization of the system. The power consumption model of LMTPS is described in Section 3.2. Our proposed policies take advantage of this non-proportionality of instantaneous power consumption to the instantaneous utilization. In our designed policies, execution of a task is almost continuous. Ignoring the migration time, the time difference between the finishing time (f_i) and start time (s_i) of any task T_i is same as the execution (or compute) time of the task, that is $(f_i - s_i) = c_i$. But in the utilization based work consolidation (UBWC), this difference is more than the given computation time for all the tasks, i.e. $(f_i - s_i) \geq c_i$.

Based on our experimental evidence, task execution with our designed scheduling policies consume less amount of overall energy on LMTMPS as compared to UBWC, even if all the tasks execute continuously. These task allocation policies are described in the following subsections.

5.1. Smart Allocation Policy (Smart)

We have designed an allocation policy called *smart allocation policy* which benefits of the non-proportionality of instantaneous system power consumption to instantaneous utilization of the system. The scheduler targets to run a set of online real time aperiodic tasks on large multi-threaded multiprocessor system under the considered power model as described in Section 3.2.

Our smart scheduling policy works on two basic ideas: (a) number of active processors should be as less as possible (active processors mean the processors which are switched on) and (b) whenever a processor is required to switch on, it should be utilized to its maximum capacity by assigning tasks to all the hardware threads of that processor if sufficient tasks are available in the system. When there is no free hardware thread in any active processor, execution of the tasks should be delayed as much as possible without missing their deadlines.

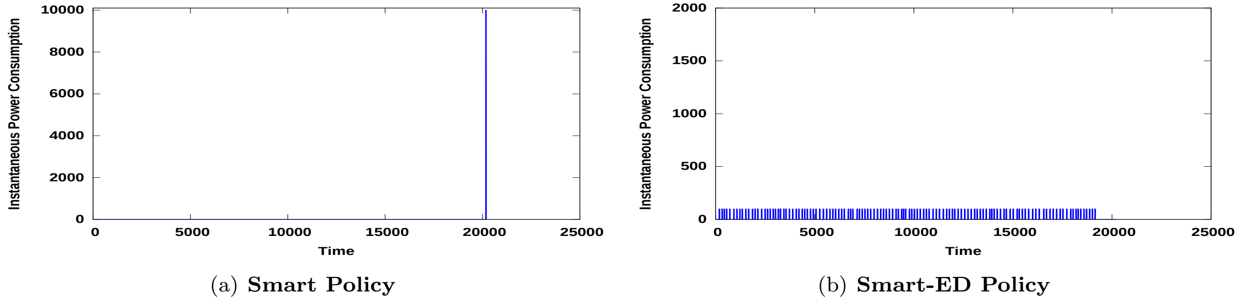
Pseudo-code of smart allocation policy is shown in Algorithm 1. In this allocation policy, whenever a task arrives to the system, it goes to waiting queue. At every time instant, our smart scheduler finds all the urgent tasks in the system. A urgent task must start its execution at current time to meet the deadline, which satisfy $(t+c_i) == d_i$, and we cannot further delay the execution of the urgent tasks. These urgent tasks are allocated to free hardware threads of active processor if exists. Otherwise new processor is switched on and new hardware threads are initiated on it to allocate such tasks. On the other hand, if there is no urgent task present in the system but there is partially filled processors, then all the hardware threads of active processors are fully utilized by allocating tasks from the waiting queue based on any policy. For simplicity we have considered first come first serve (FCFS) policy to select the tasks from the waiting queue. System information is update immediately if any task finishes its execution in a particular timestamp. The scheduler performs a consolidation operation in periodic basis. All the active hosts are sorted based their utilization (that is number of active threads). Then tasks from the least utilized hosts are migrated to the most utilized host which has a free thread. Consolidation operation is repeated such that at most one processor remains partially filled. Then all idle processors are switched off and removed from the active list. Line number 20 to 24 in the pseudo code reflects the same. The parameter *consolidationInterval* determines the frequency of consolidation operation in the scheduling process.

Figure 5 shows the plot of total power consumption verses virtual processors similar to Figure 3 with extra annotation to describe our smart allocation policy. In this case, every processor has $r = 8$ hardware threads (virtual processors) and each hardware thread consumes $\delta = 10$ amount of power. Whenever a processor is switched on, it consumes 100 units and each additional thread consumes 10 units of power. This continues till number of active hardware threads reaches $r = 8$. At this point, the processor is utilized to its maximum capacity and the total power consumption of the system is 180

Algorithm 1 Smart allocation policy

1: timestamp $t \leftarrow 0$;
2: **while** true **do**
3: **if** any task arrives, **then** Add it to Waiting Queue.
4: **while** Waiting Queue is NOT Empty **do**
5: **while** T_i = *findUrgentTask()* is not NULL **do**
6: **if** free hardware thread exists in any active processor **then**
7: allocate task T_i to a free hardware thread .
8: **else**
9: Switch on a new processor and initiate a hardware thread.
10: Allocate the task T_i to the hardware thread.
11: $s_i \leftarrow t; f_j \leftarrow t + c_i$; ▷ t is current time
12: Remove T_i from the Waiting Queue.
13: **while** there is free hardware threads in any processor **do**
14: Choose a task T_j from waiting queue based on the some policy .
15: Assign the task T_j to a free hardware thread of that processor.
16: $s_j \leftarrow t; f_j \leftarrow t + c_j$;
17: Remove T_j from the Waiting Queue.
18: **if** any task finishes its execution, **then** Update system information.
19: timestamp $t \leftarrow t + 1$;
20: **if** $(t \,\%\, consolidationInterval == 0)$ **then**
21: Sort the active processors in ascending order of their utilization.
22: Migrate tasks from least utilized processor to most utilized processor wherever free thread is there.
23: Repeat step 22 until there is at most one partially filled processor in the system.
24: Turn off the idle processors.

1: **procedure** FINDURGENTTASK()
2: **for** each task T_i in Waiting Queue **do**
3: **if** $d_i == c_i + t$ **then** ▷ t is current time
4: **return** T_i
5: **end for**
6: **return** NULL
7: **end procedure**



(a) **Smart Policy** (b) **Smart-ED Policy**

Figure 6: **Instantaneous power consumption for common deadline scheme with** $\mu = 20$, $\sigma = 10$

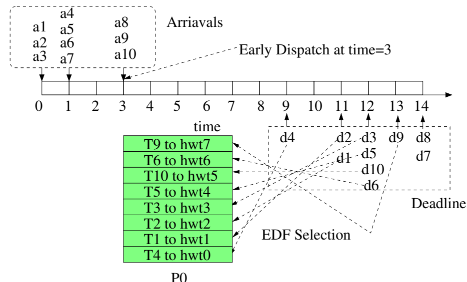


Figure 7: **Smart Allocation Policy with Early Dispatch (Smart-ED)**

(=$100 + 8 * 10$) units. The intention of the policy is to keep the system at this point if the deadline constraints of the tasks allow it to do so. As soon as the number reaches 9, another processor needs to be switched on and the total power consumption of the system increases by 110 units and reaches a total of 290 ($= 180 + 110$) units.

We define two type of points called *wait* and *go* as appeared in Figure 5. At point *wait*, the target of the policy is to delay the ready tasks as much as possible without missing any deadline. This is because, whatever processors are there in the system, they are utilized to their maximum capacity. And if another task is to be allocated, we need to switch on a new processor and the power consumption will increase and make a sharp jump to the next higher level. The other point

is called the *go* point. This indicates the point when a new processor is already switched on with only one active hardware thread. The region between the *go* point and *wait* point is called the *Filling Fast* region. In this region, the system tries to allocate as much tasks as possible either from the waiting queue or the newly coming tasks till the *wait* point is reached. As this policy uses the non-proportionality behavior of the power model of the LMTMPS, this saves a lot of energy of the system in executing the tasks. Smart is an online scheduling policy and it takes scheduling decision dynamically at runtime. Even if the actual execution time of the tasks are not known before hand and tasks do not always consume worst case execution time, the smart policy can handle this dynamic situation because mapping of tasks to the hardware threads of a processor is done irrespective to the execution time of the tasks.

5.2. Smart Allocation Policy with Early Dispatch (Smart-ED)

In smart allocation policy, initially, all the arrived tasks wait in the waiting queue till the time instant when further waiting will result a deadline miss. At this time instant, at least one task becomes urgent. We name this time instant as *urgent point*. If the urgent points for many tasks ($>> r$) get accumulated to one time instant, then we need to switch on many processors and it will create a high jump in the instantaneous power consumption value. To handle such situation we have modified our proposed smart allocation policy and the modified policy includes **early dispatch** along with the smart allocation.

The basic idea of this policy is that whenever there are tasks in the system, they need to be executed. The smart policy would make the tasks wait in the system if their deadlines permit them to wait. But this policy does not make the tasks wait till their urgent points. In this smart allocation with early dispatch policy, at any time instant a new processor is switched on if the number waiting tasks $\geq r$. After switching on the processor, r number of tasks from waiting queue is selected and are allocated to all the hardware threads of the newly switched on processor. The selection can be done using any policy.

Figure 7 shows an example of task scheduling system, where 10 tasks (with $c_i = 4$) arrived to the system on or before time 3 and earliest deadline of the tasks is at time 9. So the scheduler switches on a new processor (assuming each processor has $r = 8$ hardware threads) and schedules execution of 8 tasks (based on EDF) on to the system at time 3. Based on EDF, the selected and scheduled tasks at time 3 are T_1, T_2, T_3, T_4, T_5, T_6, T_9 and T_{10} of hardware threads $hwt0$ to $hwt7$ of processor $P0$. In this policy, at all the time instants, the scheduler selects the earliest deadline task and schedules onto an already switched-on processor if any hardware thread is free.

This policy efficiently handles the common deadline scheme where the sharp jump in the instantaneous power value near the urgent point is avoided. Instead the policy tries to equally distributes the power consumption value across time line. Figure 6 shows the instantaneous power consumption for both Smart and Smart-ED policy for the common deadline scheme. This figure clearly depicts the difference between these two policies. Smart policy shows a huge jump in the power consumption value from 0 to 12570 units at time 20155. On the other hand Smart-ED policy equally distributes the power consumption value across the time slots. This policy is preferred over the smart policy when the system is not capable of handling such high value of power consumption or a huge jump in the value. Another benefit of this policy is that it reduces the waiting time of the tasks but this is not within the scope of the paper and is not discussed here.

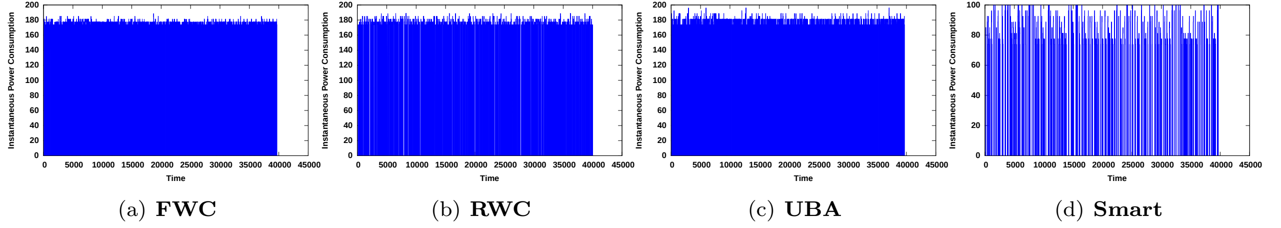
5.3. Smart Allocation Policy with Reserve Slots (Smart-R)

This is a variation of the smart allocation policy where a fraction of the processor capacity is reserved for future urgent use. In the filling fast region (as described in Figure 5), all the free hardware threads of the processor was filled with waiting tasks in basic smart policy. But in this policy, a few threads (called *Reserve Factor*) are kept free such that they can execute suddenly occurring urgent tasks (for these tasks slack time ($d_i - (c_i + a_i)$) is almost zero). This reduces the number of processors to be switched on (by compulsion) for servicing the suddenly occurring urgent tasks. This in turn might reduces the power consumption of the system. This policy will be highly beneficial for the applications where some critical tasks (having tight deadline) arrive in between regular tasks. The reserve threads in processor can execute the suddenly appearing tight deadline tasks easily.

Based on observation studied by Hsu et al. [43], power consumption of a single processor varies non-proportionally with utilization. Here the increase in power consumption with utilization is less if utilization ranges from 0 to 75%, but power consumption increases drastically if the utilization is above 75%. We can easily correlate number of active hardware threads of a processor to utilization of the processor. Suppose a processor has 8 hardware threads, means per active threads utilization is 12.5%. Number of active hardware threads of the considered 8 hardware threaded processor should be less than 6 (utilization $\leq$ 75%) as far as possible to use the processor in energy efficient manner. So power model derived in Section 3.2, can be refined further where each hardware thread of a switched on processor will not consume equal amount of power. As the number of active hardware threads crosses 75% of the processor's maximum capacity, per thread power consumption becomes higher. We believe Smart-R policy can perform even better under this power consmuption model. We leave this as a future work of this paper.

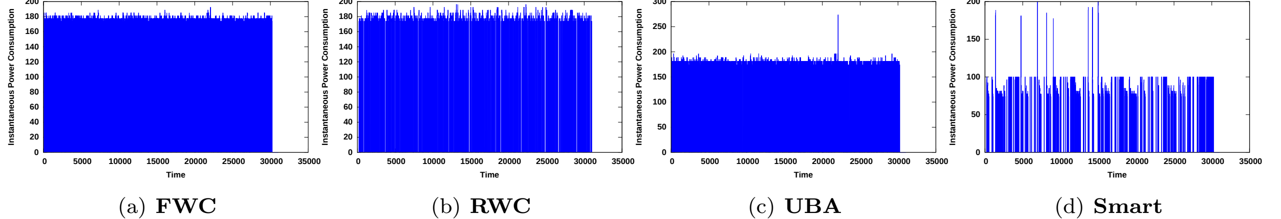
5.4. Smart Allocation Policy with Handling Immediate Urgency (Smart-HIU)

In the baseline smart allocation policy, when there is free hardware threads (represented by fast filling region in Figure 5), the policy selects some tasks from the waiting queue using FCFS in order to utilize the free hardware threads. But it might so happen that FCFS method selects tasks whose deadlines are relatively far. The near deadline tasks will eventually become urgent in near future. This urgency may force the system to start a new processor. Again it is already discussed earlier that switching on a processor by compulsion generally increases energy consumption and it is always beneficial to avoid the compulsion scenario.



(a) **FWC** (b) **RWC** (c) **UBA** (d) **Smart**

Figure 8: **Instantaneous power consumption vs time for different allocation policies under Gaussianly distributed deadline scheme ($\mu = 40$ and $\sigma = 20$)**



(a) **FWC** (b) **RWC** (c) **UBA** (d) **Smart**

Figure 9: **Instantaneous power consumption vs time for different allocation policies under random computation time scheme ($\mu = 30$ and $\sigma = 15$)**

So in this modified smart allocation policy (Smart- handling immediate urgency allocation policy), the scheduler selects and executes the tasks whose deadlines are comparatively near. That is tasks with earliest deadline (immediate urgent) from waiting queue is selected to utilize the free hardware threads in fast filling region . This in turn results in forming a long time gap between the current time and time of occurrence of next urgent point. This long time gap allows the scheduler to avoid switching on a new processor and this eventually reduces the instantaneous power consumption of the system.

6. Experiment and Results

6.1. Experimental Setup

We have created a simulation environment to simulate large multi-threaded multiprocessor system for carrying out our experiments where number of processors, number of threads per processor, base power consumption of a processor, power consumption per thread can be varied. Our simulation environment generates a wide range of output statistics, including instantaneous power consumption for all time instants/slots and the overall power consumption for different input parameters and task allocation policies. Since existing energy efficient scheduling techniques for large systems are not directly comparable to our work (to the best our knowledge), the comparison is carried out with the standard task allocation policies as discussed in Section 4.

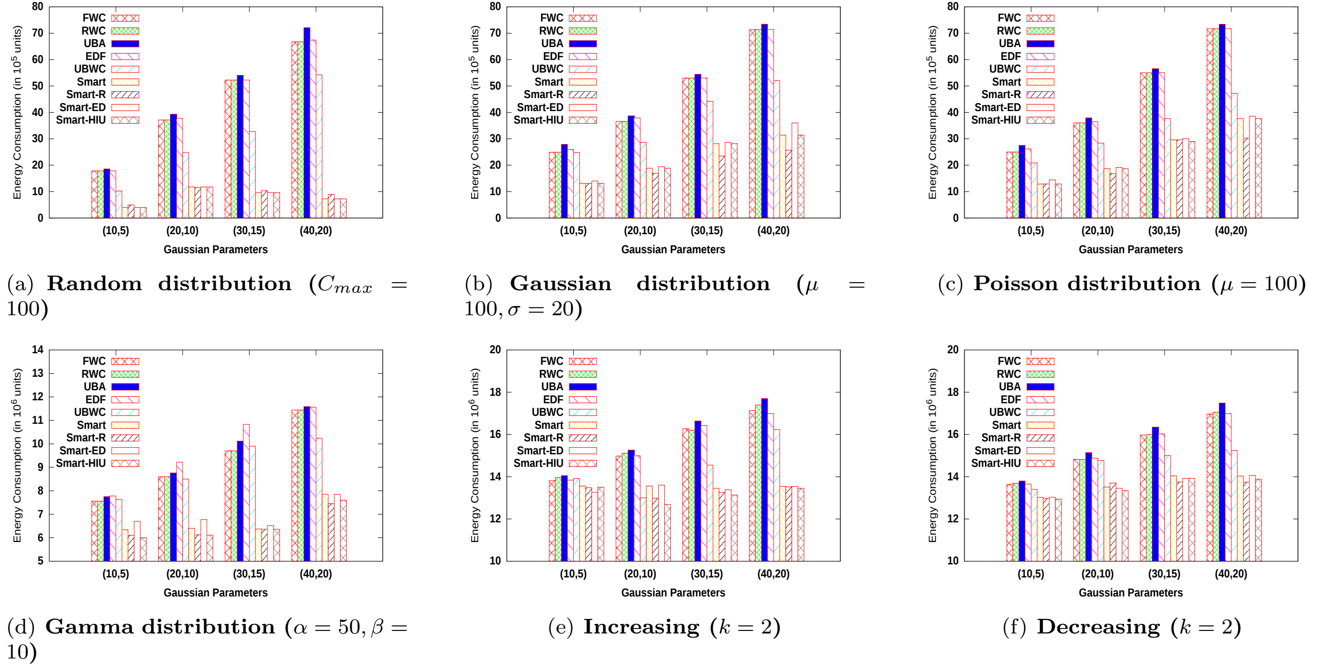
6.2. Parameter Setup

6.2.1. Processor Parameters

The static (or idle) power consumption of a processor is 70% of the total processor power consumption[44]. We have considered the total power consumption of a processor is 100 units and each processor can run up to 8 hardware threads, the base power consumption of a processor is taken as 70 units (i.e. 70% of 100) and per thread power consumption is taken as 3.75 units (i.e. $(100 - 70)/8$). We have not used the processor power consumption model described in Section 5.3.

6.2.2. Task Parameters

We have performed experiments in our simulation environment using both real workload trace data and synthetic data sets as described in Section 3.4. For synthetic data set, we considered the tasks to be online (arrives to system as time progresses), real time (have deadline), aperiodic and the inter-arrival time between consecutive tasks follows discrete Gaussian distribution. We used several pairs of (μ, σ) values e.g. $(10, 5), (20, 10), (30, 15)$ and $(40, 20)$ for generating this arrival pattern. The number of tasks generated for each experiment is 1000 and we used 10 such sets for all the cases. As stated in Section 3.4.1, we have used four different distributions and two functions for modeling the computation time of tasks. For Random $C_{max} = 100$, for Gaussian $\mu = 100, \sigma = 20$, for Poisson $\lambda = 100$, for Gamma $\alpha = 50, \beta = 10$, in Increasing $k = 2$ and for Decreasing $k = 2$. We have also considered five different types of deadline schemes. In Random, $Z_{max} = 1000$; in Gaussian, $\mu = 10, \sigma = 5$; in Increasing, $k = 3$; in Decreasing, $k = 3$; and in Common, $D = 10205$. The number of tasks generated for each experiment is 1000 and we used 10 such sets for all the cases. For our experiments, we used several pairs of (μ, σ) values e.g. $(10, 5), (20, 10), (30, 15)$ and $(40, 20)$ for generating online aperiodic tasks.



(a) **Random distribution** (C_{max} = 100)

(b) **Gaussian distribution** ($\mu = 100, \sigma = 20$)

(c) **Poisson distribution** ($\mu = 100$)

(d) **Gamma distribution** ($\alpha = 50, \beta = 10$)

(e) **Increasing** ($k = 2$)

(f) **Decreasing** ($k = 2$)

Figure 10: **Total power consumption for different allocation policies under different computation time schemes**

6.2.3. Migration Overhead

Migration overhead in a system typically depends on various factors like *total number of migrations*, *frequency of migrations*, *migration path*, *working set size*. The overhead is expressed in terms of performance degradation of the system (decrease in IPC), increase in execution time, etc. and the literature reported migration overhead as an average of 2 to 3 % and context switch overhead as less than 0.1% [33, 45]. As the thread executing a task need to run for some additional amount of time, we assume an increase of 2.5% in thread power in case of migration and 0.1% in case of preemption as the overhead. Thus as the number of migration and preemption increases, total overhead due to these also increases. As it is already mentioned in Section 4 that UBA suffers from infinite number of migrations and infinite number of preemptions, this overhead is not included while calculating the energy consumption for this policy.

6.3. Observation for Instantaneous Power Consumption

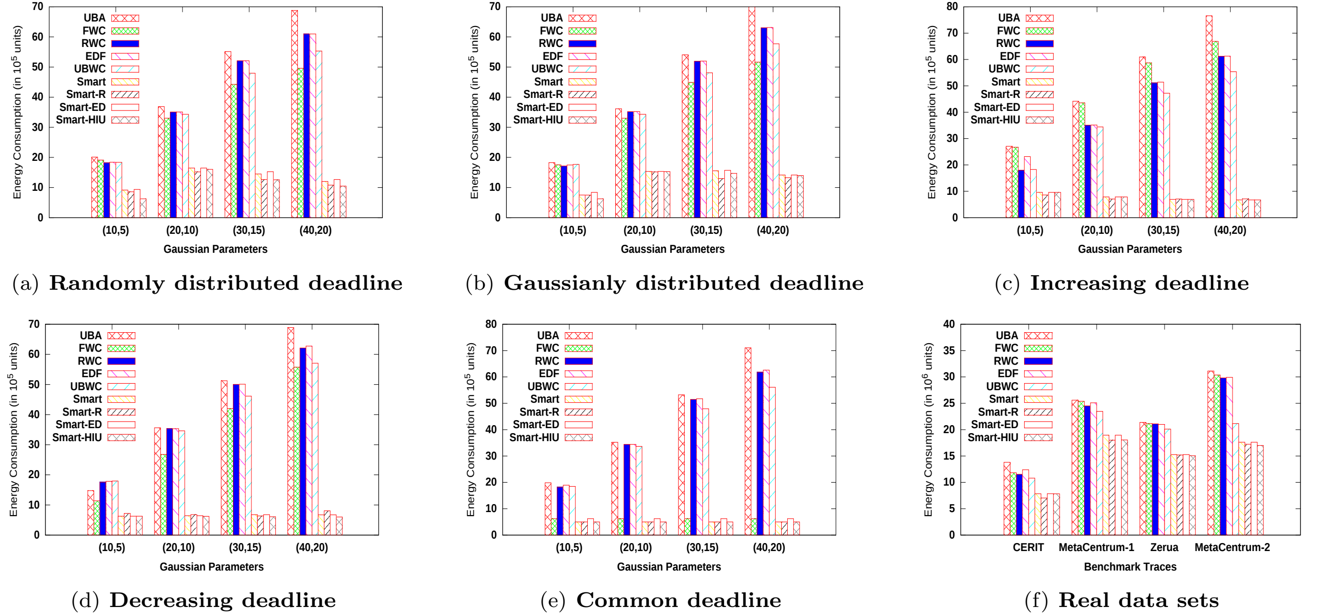
Figure 8(a) 8(b), 8(c) and 8(d) show the instantaneous power consumption of the system for FWC, RWC, UBA and Smart allocation policies under Gaussianly distributed deadline scheme respectively. Similarly Figure 9(a), 9(b), 9(c) and 9(d) show the same for random computation time scheme. We observe that our proposed policy (Smart) shows an attractive power consumption behavior as compared to others and we see many gaps in the power consumption graph (Figures 8(d) and 9(d)). These gaps indicate zero instantaneous power consumption value for that time instant. We also observe that Smart allocation policy behaves as RWC for the initial period till the occurrence of first urgent point.

6.4. Result and Analysis

Figure 10(a), 10(b), 10(c), 10(d), 10(e) and 10(f) show the energy consumption of 1000 aperiodic tasks on considered large multi-threaded multiprocessor system using different allocation policies for synthetic task set under (a) random distribution, (b) Gaussian distribution, (c) Poisson distribution, (d) Gamma distribution, (e) increasing, and (f) decreasing computation time schemes respectively. We observe that the proposed policies perform better than all other baseline policies for all the computation time schemes. Similarly, Figure 11(a), 11(b), 11(c), 11(d) and 11(e) show the total energy consumption of the system under (a) randomly distributed, (b) Gaussianly distributed, (c) increasing, (d) decreasing, and (e) common deadline schemes respectively. We can clearly observe that the energy consumption under our proposed policies is significantly lesser than all other baseline policies for all kinds of task models.

Energy consumption of the system is the summation of instantaneous power consumption over all the time slots. A low value of μ and σ indicates that the inter-arrival time of tasks are less, which signifies the system is overloaded or filled with many tasks. In such cases, opportunity to take the advantage of the smart allocation policy idea to save energy is thin. Hence the benefit is not that significant for low values of μ and σ. As the values of μ and σ increase, the inter-arrival time between tasks increases. Thus the system becomes loosely loaded and smart allocation takes the benefit of it. More sparse the tasks are, more reduction in energy by smart allocation policy as compared to others.

The experimental results also establishes our claim that Smart-R and Smart-HIU can further reduce the energy consumption of the system almost in all the cases. But the total energy consumption for Smart-ED is same or little higher than Smart. This is justified as the purpose of the Smart-ED policy is to avoid the sharp jump in the power consumption value and this policy distributes the power value across the time axis. This was explained in Section in 5.2 with an example. The proposed policy achieves average energy reduction of 37% as compared to others for different computation time schemes



(a) **Randomly distributed deadline** (b) **Gaussianly distributed deadline** (c) **Increasing deadline**

(d) **Decreasing deadline** (e) **Common deadline** (f) **Real data sets**

Figure 11: **Total power consumption for different allocation policies for both synthetic and real data sets**

and of 55% as compared to others for different deadline schemes. Based on this experimental result, we can firmly say that our proposed policies reduce energy consumption of the system significantly for all of the cases of synthetic workload.

Table 1 shows the number of migrations with different task allocation policies both for synthetic and real trace data. As already explained earlier, UBA is of theoretical interest and has an unbounded number of migrations. FWC and RWC are of non-preemptive nature and do not require any migration but they lack in the overall energy consumption with respect to the proposed policies. We have considered a non-preemptive implementation of EDF and thus the number of migrations in this case is also 0. We observe that total number of migrations in all our proposed policies are within a reasonable range.

6.5. *Experiments with Real Workload Traces*

In order to validate our proposed work, we have performed the experiments with real workload traces. The experiments were performed for four different workloads: CERIT-SC, MetaCentrum-1, Zewura and MetaCentrum-2 data sets. CERIT-SC and Zewura contains job descriptions of 17900 jobs (same as tasks) and 17256 jobs respectively. We have considered execution of all the jobs for both the CERIT-SC and Zewura workload traces. On the other hand MetaCentrum-1 and MetaCentrum-2 work load traces contains job descriptions of 495299 jobs and 103656 job respectively. Since it is relatively difficult to conduct experiments with such large numbers, we have taken first 15000 jobs in our experiments. All the workload traces contain the job id (i.e task number), arrival time, computation time along with other information. Deadline of a job is taken as *arrival time* + *computation time* + z; where z is a random number, varies in the range 0 to 1000.

Figure 11(f) shows the energy consumption of real workload traces (CERIT-SC, MetaCentrum-1, Zewura and MetaCentrum-2) on considered large multi-threaded multiprocessor system using different allocation policies. Our proposed policies outperform the rests. Our proposed policies achieve maximum energy reduction up to 44% as compared to all the earlier allocation policies UBA, WFC, RWC, EDF and UBWC. The proposed policies achieve average energy reduction of 30% as compared to the baseline policies. Experimental results show that the energy reduction in case of real trace data is comparatively lesser than that of synthetic data. This is because the inter-arrival time of tasks in case of real trace data is less and the smart policy can not take the benefit of completely switching off the processors for longer time.

Energy consumption in case of UBWC is in general lesser than that of UBA, FWC and RWC but this policy incurs significant number of migrations. It can be lucidly seen from the experimental data that our proposed policies not only reduces overall energy consumption by a significant margin, but also the number of migrations in these policies are within reasonable range. Thus it can be concluded that even in case of high migration overhead system, the proposed policies will achieve a significant energy reduction.

7. Conclusion and Future work

Energy aware scheduling at a coarser granularity level has become essential for the large multi-threaded multiprocessor systems. In this paper, we have derived a simple power consumption model for such large systems and proposed an online energy efficient task allocation policy, namely, *smart allocation policy* for executing a set of independent real time tasks. We have then proposed three variations of this policy to further reduce energy consumption (for some applications) and to efficiently handle different situations which might occur at runtime. Experimental result reveals that our proposed policies perform significantly better than all other five baseline policies both for synthetic data and real workload traces.

In near future, we will be considering scheduling tasks with dependencies. Efficient scheduling of multiprocessor tasks (tasks requires more than one thread for execution) under this power model will be a great extension to this work.

Scheduling policies →	UBA	FWC	RWC	UBWC	EDF	Smart	Smart-ED	Smart-R	Smart-HIU
Data sets ↓	-	-	-	-	-	-	-	-	-
Random computation time	∞	0	0	490	0	7	2	9	7
Gaussian computation time	∞	0	0	624	0	17	27	32	15
Poisson computation time	∞	0	0	527	0	10	16	28	9
Gamma computation time	∞	0	0	37421	0	142	204	92	97
Increasing computation time	∞	0	0	14542	0	124	102	99	112
Decreasing computation time	∞	0	0	14221	0	112	90	87	120
Random deadline time	∞	0	0	442	0	4	0	2	2
Gaussian deadline time	∞	0	0	538	0	9	0	1	1
Increasing deadline time	∞	0	0	342	0	3	3	2	8
Decreasing deadline time	∞	0	0	382	0	12	9	15	17
Common deadline time	∞	0	0	702	0	24	21	19	17
Real trace data	∞	0	0	29908	0	240	196	29	24

Table 1: **Number of migrations occurred in different scheduling policies for different data sets**

Extending the simple power consumption model to support multilevel clustered architectures will help the community to use this work for general purpose. Also in future, we will consider processor, memory and I/O utilization of tasks to schedule them efficiently. Scheduling real time tasks with Smart-R policy can be more attractive under the power consumption model as discussed in [43]. We have kept this as an immediate extension of our current work.

8. Acknowledgement

The authors of the paper would like to thank Prof. Ce-Kuen Shieh and anonymous reviewers for reviewing this manuscript and forwarding their constructive comments. This has greatly improved the quality of this paper.

References

[1] Robert I. Davis and Alan Burns. A Survey of Hard Real-time Scheduling for Multiprocessor Systems. *ACM Computing Survey*, pages 35:1–35:44, October 2011.

[2] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Real-Time Systems Series. Springer, 2011.

[3] C. Hankendi and A.K. Coskun. Adaptive Power and Resource Management Techniques for Multi-threaded Workloads. In *IEEE 27th IPDPSW*, pages 2302–2305, May 2013.

[4] Wan Yeon Lee. Energy-Efficient Scheduling of Periodic Real-Time Tasks on Lightly Loaded Multicore Processors. *IEEE TPDS*, pages 530–537, March 2012.

[5] S. Zhuravlev et al. Survey of Energy-Cognizant Scheduling Techniques. *IEEE TPDS*, pages 1447–1464, July 2013.

[6] Karel De Vogeleer et al. The Energy/Frequency Convexity Rule: Modeling and Experimental Validation on Mobile Devices. In *10th Int. Conf. Para. Processing and Applied Mathematics*, pages 793–803, 2013.

[7] T. D. Burd and R. W. Brodersen. Energy Efficient CMOS Microprocessor Design. In *Proceed. of the 28th Hawaii Int. Conf. on System Sciences*, HICSS, 1995.

[8] M.B. Taylor. A Landscape of the New Dark Silicon Design Regime. *IEEE/ACM MICRO*, Sept 2013.

[9] J.M. Allred et al. Dark Silicon Aware Multicore Systems: Employing Design Automation With Architectural Insight. *IEEE Trans. on VLSI Systems*, May 2014.

[10] X. Zhu et al. Real-Time Tasks Oriented Energy-Aware Scheduling in Virtualized Clouds. *IEEE TCC*, pages 168–180, April 2014.

[11] Dong Jiankang, , et al. Virtual Machine Scheduling for Improving Energy Efficiency in IaaS Cloud. *Communications, China*, pages 1–12, March 2014.

[12] Power consumption tests. http://www.xbitlabs.com/.

[13] Power consumption qualcom hexagon v3. http://www.bdti.com/.

[14] Canturk Isci et al. Agile, Efficient Virtualization Power Management with Low-Latency Server Power States. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 96–107, 2013.

[15] Chao Xu et al. Automated OS-level Device Runtime Power Management. In *20th Int. Conf. on Architectural Support for Prog. Lang. and Op. Sys.*, ASPLOS '15, pages 239–252, 2015.

[16] Mark Weiser et al. Scheduling for Reduced CPU Energy. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94.

[17] H. Aydin et al. Dynamic and Aggressive Scheduling Techniques for Power-Aware Real-Time Systems. In *22nd IEEE RTSS*, pages 95–105, Dec 2001.

[18] H. Aydin et al. Optimal Reward-Based Scheduling for Periodic Real-Time Tasks. *IEEE TC*, 50(2):111–130, Feb 2001.

[19] D. Zhu et al. Scheduling with Dynamic Voltage/Speed Adjustment using Slack Reclamation in Multiprocessor Real-Time Systems. *IEEE TPDS*, pages 686–700, July 2003.

[20] C. Isci et al. An Analysis of Efficient Multi-Core Global Power Management Policies: Maximizing Performance for a Given Power Budget. In *39th MICRO*, 2006.

[21] Young Choon Lee and Albert Y. Zomaya. Minimizing Energy Consumption for Precedence-Constrained Applications Using Dynamic Voltage Scaling. In *9th IEEE/ACM Int. Symp. on CCGrid*, 2009.

[22] Young Choon Lee and A.Y. Zomaya. Energy Conscious Scheduling for Distributed Computing Systems under Different Operating Conditions. *IEEE TPDS*, 2011.

[23] Dawei Li and Jie Wu. Energy-Aware Scheduling for Aperiodic Tasks on Multi-core Processors. In *43rd ICPP*, pages 361–370, 2014.

[24] Dawei Li and Jie Wu. Minimizing Energy Consumption for Frame-Based Tasks on Heterogeneous Multiprocessor Platforms. *IEEE TPDS*, 26(3):810–823, 2015.

[25] Dawei Li and Jie Wu. Energy-Aware Scheduling for Frame-Based Tasks on Heterogeneous Multiprocessor Platforms. In *41st ICPP*, pages 430–439, 2012.

[26] Jeffrey S. Chase and Ronald P. Doyle. Balance of Power: Energy Management for Server Clusters. In *HotOS*, 2001.

[27] Jeffrey S. Chase et al. Managing Energy and Server Resources in Hosting Centers. *SIGOPS Operating System Reviews*, 35(5), 2001.

[28] Y. Hotta et al. Profile-based Optimization of Power Performance by using Dynamic Voltage Scaling on a PC Cluster. In *20th Int. Sympo. on Para. and Distri. Proc.*, April 2006.

[29] S. Srikantaiah et al. . In *Proceedings of Conference on Power Aware Computing and Systems*, HotPower, 2008.

[30] P. Kokkinos et al. Data Consolidation: A Task Scheduling and Data Migration Technique for Grid Networks. In *8th IEEE Int. Sympo. on CCGRID*, pages 722–727, 2008.

[31] Jeonghwan Choi et al. Power Consumption Prediction and Power-Aware Packing in Consolidated Environments. *IEEE TC*, 59(12):1640–1654, 2010.

[32] A. Verma et al. Power-aware Dynamic Placement of HPC Applications. In *22nd Annual Int. Conf. on Supercomputing*, ICS, pages 175–184, 2008.

[33] A. Verma et al. pMapper: Power and Migration Cost Aware Application Placement in Virtualized Systems. In *9th ACM/I-FIP/USENIX Int. Conf. on Middleware*, Middleware '08, pages 243–264, 2008.

[34] Yongqiang Gao et al. Quality of service aware power management for virtualized data centers. *Journal of Systems Architecture*, 59(4–5):245 – 259, 2013.

[35] S. Ali et al. Task execution time modeling for heterogeneous computing systems. In *Proc. of Heterogeneous Computing Workshop*, pages 185–199, 2000.

[36] Jong Kim and K. G. Shin. Execution time analysis of communicating tasks in distributed systems. *IEEE TC*, 45(5):572–579, May 1996.

[37] Peter Brucker. *Scheduling Algorithms*. Springer Publishing Company, Incorporated, 5th edition, 2010.

[38] Metacentrum data sets. https://www.fi.muni.cz/ xklusac/index.php?page=meta2009.

[39] The Standard Workload Format. http://www.cs.huji.ac.il/ labs/parallel/workload/swf.html.

[40] M. Zaharia et al. Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling. In *Pro. of the 5th European Conf. on Computer Sys.*, EuroSys '10, pages 265–278, 2010.

[41] Y. Guo et al. Preference-oriented Real-time Scheduling and Its Application in Fault-tolerant Systems. *Journal of System Architecture*, 61(2):127–139, February 2015.

[42] Anton Beloglazov and others. Energy-Aware Resource Allocation Heuristics for Efficient Management of Data Centers for Cloud Computing . *Future Generation Computer Systems*, 28(5):755 – 768, 2012.

[43] Ching-Hsien Hsu et al. Optimizing Energy Consumption with Task Consolidation in Clouds. *Information Sciences*, 258:452 – 462, 2014.

[44] Yan Maa et al. Energy-Efficient Deadline Scheduling for Heterogeneous Systems. *J. of Para. and Dist. Comp.*, 72(12):1725 – 1740, 2012.

[45] S. Holmbacka et al. Task migration for dynamic power and performance characteristics on many-core distributed operating systems. In *21st Euromicro Int. Conf. on Para., Dist., and Network-Based Processing*, pages 310–317, Feb 2013.