AN ENERGY-AWARE MANAGEMENT TECHNIQUE FOR REAL-TIME MULTI-CORE SYSTEMS

A Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Fang Liu

December 2013

AN ENERGY-AWARE MANAGEMENT TECHNIQUE FOR REAL-TIME MULTI-CORE SYSTEMS

Fang Liu

APPROVED:

Albert M.K. Cheng
Dept. of Computer Science

Weidong Shi
Dept. of Computer Science

Xiaojing Yuan
Dept. of Engineering Technology

Dean, College of Natural Sciences and Mathematics

Acknowledgements

I am thankful to my advisor, Dr. Albert M.K. Cheng. This thesis could not have been written in the current form without his lots of comments.

I'd like to thank Dr. Weidong Shi and Dr. Xiaojing Yuan for serving as my committee members. I also want to thank them for letting my defense be an enjoyable moment and for their brilliant comments.

Special gratitude also goes to my wonderful colleagues, who supported and helped me along the way. We shared the pain, but we believed that we all would gain in the end.

At last, I thank my family and my sister Bo for their unwavering support, understanding, and encouragement, both during graduate school and the years before. I could not have finished this without you.

AN ENERGY-AWARE MANAGEMENT TECHNIQUE FOR REAL-TIME MULTI-CORE SYSTEMS

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Fang Liu

December 2013

Abstract

In this work, we propose a new dynamic migration (DM) heuristic method integrating dynamic voltage scaling (DVS), dynamic power management (DPM) and task migration in multi-core real-time systems which can feasibly balance the task load and reduce energy consumption during execution to achieve energy efficiency. Meanwhile, voltage scaling-based dynamic core scaling (VSDCS) is presented for reducing leakage power consumption under low task load conditions. The framework used for the proposed methods is composed of a partitioner, a local earliest deadline first (EDF) scheduler, a power-aware manager, a dynamic migration module, and a dynamic core scaling module. The primary unit is the power-aware manager which controls the frequency for the power consumption and the voltage scaling based on the feedback of the dynamic migration module and the dynamic core scaling module.

Simulation results show that the DM heuristic can produce further energy savings of about 3 percent compared with the closest previous work [44]. That is $(1 - (1 - 8\%) \cdot (1 - 3\%)) = 11\%$ energy saved with the new DM techniques. This work also greatly reduces the cost of task migration among the multi-core processors. The results show that VSDCS can achieve up to 33 percent of energy savings under low load conditions as compared with previous methods.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The increasing computational demand is presently satisfied by using a large number of the cores on a single chip. For example, Villalpando et al. [47] evaluated the Tilera 64-core TILE64 processor which performed well in the real-time hazard detection and avoidance system of the Altair Lunar Lander. Maestro has been developed with the explicit purpose of enabling space-borne, embedded multi-core real-time systems. It is a radiation-hardened 49-core version of the TILE64 processor [35]. With the recent technological advances in multi-core chips, the power consumption has become an important design issue in modern multi-core processors. This fact is even more critical with the growth of mobile and portable platforms/devices and embedded systems.

Much research is being conducted to improve the performance of the chemical batteries while reducing energy consumption of the computer systems to extend battery lifetime. Power management plays an important role in the operation of mobile systems and embedded systems to prolong battery lifetime and reduce the increasing power consumption. The power consumption of a device is generally classified as: (1) Dynamic power consumption which arises due to the execution of instructions [17] [33]; or (2) Leakage power consumption or static power consumption which is proportional to leakage current of CMOS [25]. In order to reduce energy consumption, lowering the supply voltage V_{dd} is one of the most effective ways, since energy consumption E of CMOS circuits has a quadratic dependency on the supply voltage V_{dd} [34]. Frequency and voltage are two related elements. In other words, when lowering the frequency of the processor, the processor becomes much more stable. Dynamic voltage and frequency scaling (DVFS), dynamic voltage scaling (DVS), and dynamic power management (DPM) are the techniques to change the frequency and/or operating voltage of processors in terms of system computational demands at a given point of time [34]. The voltage increase or decrease can be used as a factor on computer operating systems, such that DVS/DVFS and DPM can be used as a power management technique from the system point of view. Voltage and frequency scaling are often used together to save power in mobile devices, embedded systems, and the real-time applications. By adopting energy-efficiency DVS CPUs and power-aware subsystems [42], a modern processor may operate at different supply voltages. Thus, the frequency (speed) of the processor can change accordingly. Intel StrongARM SA1100 processor [6] and Intel XScale [7] are such well-known DVS

processors in embedded systems.

In general, partitioned scheduling assigns the tasks statically to the processors by a certain partitioning heuristic algorithm. Each processor is scheduled individually using a uniprocessor scheduling policy such as earliest deadline first (EDF) or rate-monotonic (RM). Global scheduling occurs when all the tasks are scheduled in a queue. In a multi-core system, the DVFS regulator can work with either global states on the cores or local states on the cores. As a result, DVFS regulator can properly adjust the frequency of the cores in each partition to satisfy its application requirements. Simultaneously, the power consumption can be reduced by loading the workload evenly [22] [36] on multi-core systems with a global DVFS regulator. In this case, an extra module is required to distribute the workload among local partitions according to a given algorithm. In order to save energy, the system allows task migration from one core to another, which results in energy savings as well as workload balancing.

1.2 Contributions

The contributions of this work are briefly outlined as follows:
First, we introduced an energy-efficient management system model that exploits DVS/DVFS and DPM to save both dynamic and leakage power consumption. This system model includes a partitioner, a local EDF scheduler, a power-aware manager, a dynamic migration module, and a dynamic core scaling module. The primary module is the power aware manager. It utilizes DVS/DVFS and DPM to control

the power consumption and the voltage scaling with the feedback of the dynamic migration and the dynamic core scaling modules. This system model applies to a periodic real-time system on multi-core processors when it combines with the frequency settings of the underlying hardware.

Second, we extended dynamic repartitioning into dynamic migration (DM) [44]. Our voltage scaling-based dynamic core scaling (VSDCS) utilizes the critical frequency to determine the number of the cores instead of calculating the complicated function during run-time execution [44]. Compared with Euiseong Seo's work [44], DM further reduces the dynamic power consumption by reducing the total number of task migrations as explained in section 4.1 and 4.2. VSDCS further decreases the leakage power consumption by avoiding the non-negligible overhead as explained in section 4.3 and 4.4.

The third contribution of this study is that the algorithms can be applied for scheduling CPU-intensive applications on heterogeneous multi-core systems. Future work intends to improve the algorithms for both CPU-intensive and data-intensive applications by considering memory resources in the algorithms.

1.3 Organization

The rest of this thesis is organized as follows. Chapter 2 introduces related work and existing techniques. Chapter 3 describes a power model and a task set model used for the energy-efficient management system model. Additionally, it discusses an energy-efficient management system model. Chapter 4 describes the DM heuristic

algorithm and VSDCS heuristic algorithm which efficiently reduce dynamic power consumption and leakage power consumption. Chapter 5 analyzes simulation results for the above two algorithms. Chapter 6 presents the future directions.

Chapter 2

Related Work and Background

2.1 Related Work

Partitioned scheduling is a commonly used multiprocessor real-time scheduling approach [21]. The bin-packing problem implies that finding an optimal task assignment in all cases is an non-deterministic polynomial-time (NP) hard problem [21]. In [10], the authors have stated that worst fit (WF) is the best partitioning heuristic approach in terms of the energy efficiency. There has been much research work [41] [15] [46] [30] about using dynamic voltage scaling (DVS) on a unicore processor. The partitioning approach has the advantage of utilizing DVS. However, the use of each partitioned set should be well balanced to maximize the energy efficiency [16]. Donald et al. [22] stated that if the workload is properly balanced among the cores, global DVS can be as efficient as local DVS. Also, several commercialized processors, such as the IBM Power 7 [28], have implemented global DVS. After partitioned scheduling,

the existing DVS techniques for unicore processors are easily accomplished on each processor [41] [15]. EDF is the optimal algorithm for preempted periodic real-time task scheduling [32]. Pillai and Shin [41] proposed three DVS scheduling heuristics: static, cycle-conserving and look-ahead to conserve energy in a system requiring real-time deadline guarantees. Cycle-conserving DVS for EDF is also implemented using actual measurements and has provided significant energy savings in the real world [41]. Yang et al. [48] presented a DVS-enabled chip multiprocessor (DVSCMP), and suggested a heuristic algorithm for scheduling a framed, periodic, real-time task model, since they proved that the energy efficient scheduling of the periodic real-time tasks on DVSCMP system is an NP-hard problem. Based on cycle-conserving DVS, Euiseong Seo et al., [44] suggested two algorithms: dynamic repartitioning and dynamic core scaling for reducing both dynamic and leakage power consumption. Dynamic repartitioning algorithm is based on existing partitioning approaches for multiprocessor systems to dynamically balance the task loads among the cores based on the requirements of the computational demands. It can minimize the power consumption during execution. In addition, dynamic core scaling algorithm coupled with dynamic repartitioning adjusts the number of active cores to reduce leakage power consumption when the task load on each core is low.

2.2 Partitioned Scheduling Heuristics

Partitioned scheduling on multiprocessors is known to be a NP-Hard problem [48], since solving the task allocation problem on multiprocessors is equivalent to a bin-packing problem. Partitioned scheduling can be implemented by applying existing uniprocessor techniques on each partition (i.e., each core). In comparison with global scheduling, partitioned scheduling has its advantages. Partitioned scheduling uses a separate run-queue for each processor rather than a single global queue. One task can only affect the other tasks on the same processor if a task exceeds its worst-case execution time budget. The key of partitioned scheduling is to ensure the work load is balanced on each core. Then, assigning the tasks to a multi-core system can be converted to assign the tasks to m simpler single core problems.

Some research [21] [40] [19] has adopted EDF or fixed-priority scheduling using RM combined with bin packing heuristics including first fit (FF), next Fit (NF), best fit (BF), and worst fit (WF) in partitioned multiprocessor scheduling. The worst-case utilization bound of partitioned scheduling for periodic task sets with implicit deadlines is given in Formula (2.1) [12]:

$$U = (m + 1)/2 \tag{2.1}$$

where m is the number of processors/cores.

2.2.1 Bin-packing

The bin-packing problem is a classic NP-hard problem [48]. The problem is to assign a set of n items x_1,...x_n to a bin with capacity V so that there is no bin that has a capacity totaling more than one and the number of bins used is minimized. In the context of partitioned scheduling, a set of items is a task set and the size of the capacity is given by the utilizations. Each processor/core has a capacity of 1.0 under the conditions of implicit deadlines and EDF. Partitioning a task set is equivalent to solving a bin-packing problem. Thus finding an optimal task assignment by partitioned scheduling is intractable (unless P = NP). We can use existing bin-packing heuristics to find valid task assignments, which is a practical method. In order to describe the heuristic algorithms below, we define the set of items as Formula (2.2):

$$S_j = x_1, ..., x_n \tag{2.2}$$

which are assigned to the bin of a number of j. Here, items x_1,...x_n are with corresponding sizes y_1, ..., y_n. Then, we define the remaining capacity of the jth bin is V_j [27]. That is,

$$V_j = V - \sum_{x_i \in S_j} y_i \tag{2.3}$$

At the beginning, there is only one empty bin, but when the capacity of the existing bins is not sufficient to use, additional bins can be added according to the heuristic. In this work, items are sorted in the order of decreasing sizes in (2.4):

$$y_1 \geq y_2 \geq ... \geq y_n \tag{2.4}$$

We used first fit decreasing (FFD), next fit decreasing (NFD), best fit decreasing (BFD), and worst fit decreasing (WFD) heuristic algorithms for partitioned scheduling.

First Fit Decreasing

Assign item x_i to the first bin into which it will fit. That is, item x_i is placed to S_j if it fits, then [27]:

$$V_j + y_i \leq 1 \tag{2.5}$$

If no such j exists, x_i will be placed in a new empty bin.

Best Fit Decreasing

Assign x_i into a bin to make the best of usage of the bin into which it goes. That means x_i goes to the bin with the lowest remaining capacity. So, if it fits, then [27]:

$$let\ L = \{l | V_l \geq y_i\},\ x_i \in S_j = min\{j \in l | V_j + y_j \leq 1\} \tag{2.6}$$

BFD maximizes the usage of the bin. If more than one bin satisfies (2.6), the left-most one will be chosen.

Worst Fit Decreasing

Assign x_i into the bin with the most remaining capacity. Otherwise, x_i will be put in the left-most empty bin. The worst fit heuristic is the inverse of the best fit heuristic, then [27]:

$$let\ L = \{l | V_l \geq y_i\},\ x_i \in S_j = max\{j \in l | V_j + y_j \leq 1\} \tag{2.7}$$

If more than one bin satisfies (2.7), the left-most one will be chosen.

Next Fit Decreasing

When processing the next item, see if it fits in the same bin as the last item. Start a new bin only if it does not.

There is a slight difference between the problem of partitioning a task set onto a fixed number of the cores on a multi-core platform and the bin-packing problem. Since there are initially m empty bins, it is not possible to allocate additional bins to the task set. This benefits the worst fit heuristic because WFD attempts to allocate the total utilization evenly among all the cores. Instead, FFD, BFD, and NFD heuristics attempt to fully allocate a core before assigning to the next core. Assigning the load among all the cores is preferable under partitioned scheduling on multi-cores. Thus, WFD is the best heuristic among the above for our work.

2.2.2 A Case Study

In the case study, 7 tasks are sorted in decreasing order. The sizes of the tasks are given as the following: 0.45, 0.35, 0.28, 0.25, 0.17, 0.09, and 0.01. There are three empty cores. Due to its size y_i and the selected bin-packing heuristic, the next task x_i will be put into the fit core, as indicated in Figure 3.2. At first, x_1 arrives with $y_1 = 0.45$. Then for all heuristic algorithms, x_1 will be assigned to S_1. When x_2 with $y_2 = 0.35$ comes in, WFD will assign x_2 to the next core S_2 with the most remaining capacity. However, BFD will assign x_2 to S_1 with the least remaining capacity that is still sufficient to fit for. FFD will assign x_2 to S_1 which is the first fit core. NFD will assign x_2 to S_1 which is the same bin as the last item.

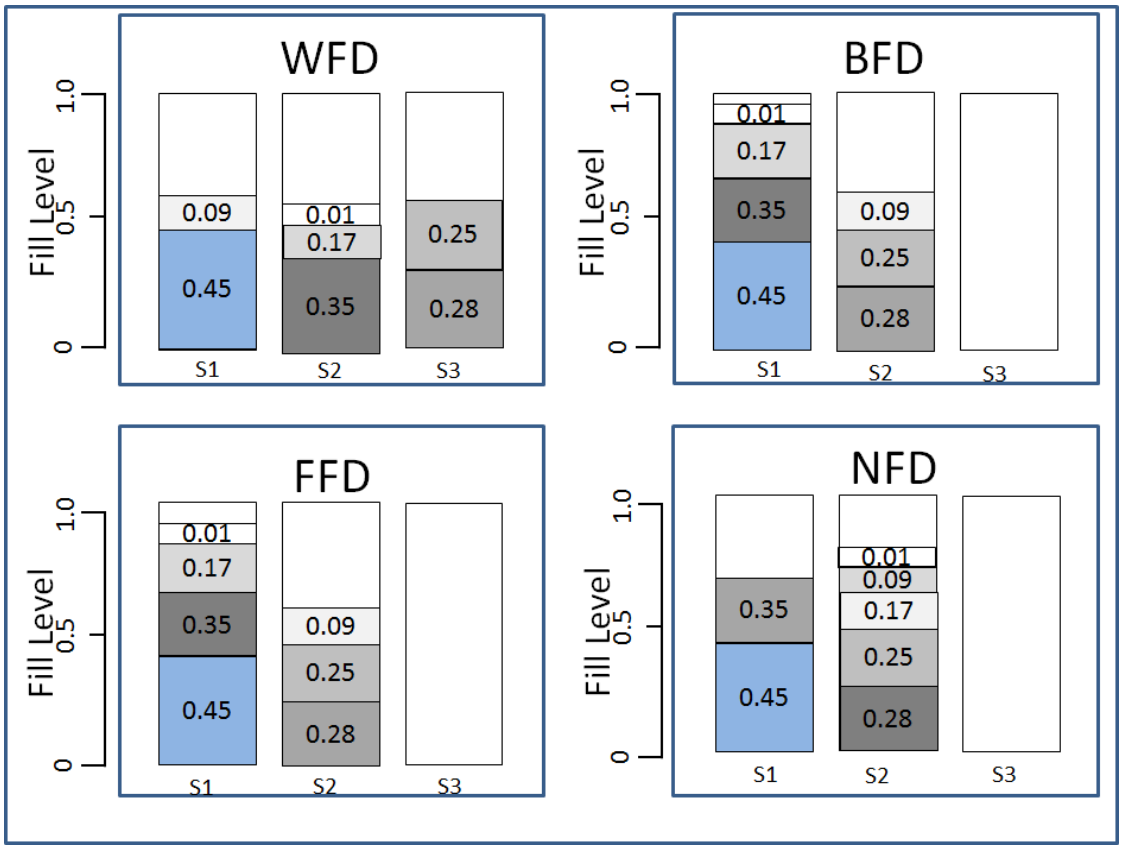


Figure 2.1: A Case Study of WFD, BFD, FFD, and NFD

2.3 Cycle-conserving DVS for EDF

2.3.1 Cycle-conserving DVS for EDF

EDF can guarantee that all the tasks will meet the real-time constraints when the total utilization of all the tasks is less than one. For preemptible periodic real-time task scheduling, EDF is the optimal algorithm. Based on EDF, Pillai and Shin [41] proposed a cycle-conserving DVS scheduling heuristic for dynamic power aware scheduling on uniprocessors. Cycle-conserving avoids wasting cycles by taking advantage of surplus time to run the other remaining tasks at a lower CPU frequency rather than accomplish more work, e.g. slack time stealing [12]. When the task completes, the actual processor cycles used by the tasks may be less than the worst-case specification. Any unused cycles that were allotted to the task would be wasted. Therefore, cycle-conserving reclaims the extra time by recaculating utilization using the actual computation time consumed by the task, when a task completes earlier than its worst-case computation time. This lower operating frequency is used until the task is released again for its next invocation.

The cycle-conserving algorithm works as follows:

The total utilization of the tasks is initialized with the utilization calculated with worst case execution time (WCET). The value of each task's utilization is updated both when the task is scheduled and when the task is finished. Upon completion of a task, the task utilization is updated based on the task's actual execution time. On the release of the task before execution, its utilization is restored to the original value based on the task's WCET. The cycle-conserving algorithm

Task	Period	WCET	Utilization	cc_1	cc_2
τ_1	10ms	3ms	0.3	2ms	1ms
τ_2	14ms	4ms	0.286	2ms	1ms
τ_3	15ms	3ms	0.2	1ms	1ms

Table 2.1: Example Task Set

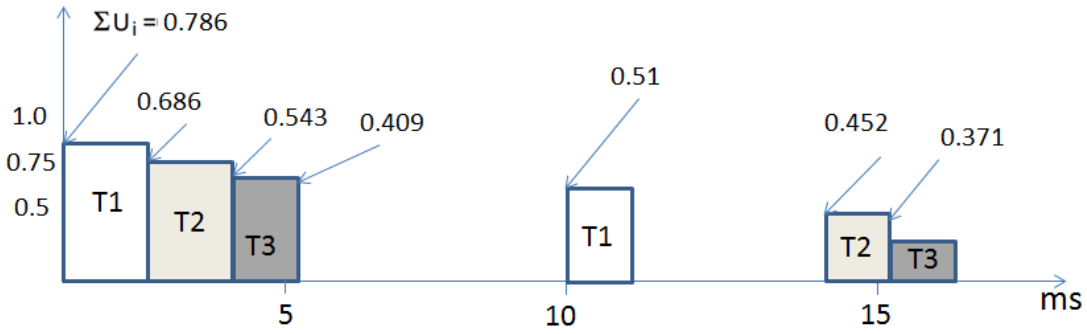


Figure 2.2: A Case Study of Cycle-conserving DVS for EDF

may change the frequency with a fraction of the maximum frequency at the point when it completes. In this manner, the cycle-conserving algorithm may obtain the reduced frequency during the period between a task's completion and the start of its next invocation. Then, a conservative assumption must be made that it will need its worst case computation time when the tasks are rescheduled according to the EDF schedulability test. At this point, the deadline guarantees are maintained.

2.3.2 A Case Study

Figure 2.2 shows an example of the cycle-conserving algorithm. Table 2.1 lists the actual execution time cc_1 and cc_2, WCET, and period (deadline) of the tasks τ_1, τ_2, and τ_3. The actual execution time of instance 1 of τ_1 is 2ms. Then, the total utilization of τ_1, τ_2, and τ_3 is thus updated from 0.786 to 0.686 after τ_1 is completed.

At this time, the core will be operated at 0.686 times the highest frequency. In this manner, after the instance 1 of τ_2 is completed, the total utilization decreases to 0.543 since the actual execution time of τ_2 is 2ms shorter than WCET of τ_2. Thus, the remaining task τ_3 will be operating at 0.543 times the highest frequency. When the first instances of τ_1, τ_2, and τ_3 are completed, T_1 begins executing in the next period. Then the utilization of τ_1 is restored to the original $WCET/period$. The utilizations of τ_2 and τ_3 reserve *the actual execution time/period*.

Chapter 3

An Energy-efficient Management System on Multi-core Processors

In this chapter, we first propose a power-aware model, which has been generally used in energy-efficient research. Then we discuss a task set model for the periodic tasks on multi-core systems. In the end, we discuss an energy-efficient management system to improve the trade-off between energy conservation and load balancing.

3.1 Power Consumption Model

There are two kinds of energy consumption on ICs: dynamic power consumption and leakage power consumption [17] [33]. Dynamic power or switching power is the result of charge and subsequent discharge of digital circuits. Leakage power or static power is the result of leakage current in CMOS circuits.

3.1.1 Dynamic Power Consumption

The dynamic power consumption can be calculated using (3.1) [17] [33][20]:

$$P_{dynamic} = \sum_{k=1}^{N} C_k \cdot f_k \cdot V^2 \tag{3.1}$$

where V is the supply voltage, N is the number of gates in a circuit, C_k is the load capacitance of gate and f is the switching frequency of gate. Let

$$\sum_{k=1}^{N} C_k \cdot f_k \cdot V^2 = C \cdot f \cdot V^2 \tag{3.2}$$

Then, $P_{dynamic}$ can be expressed as (3.3):

$$P_{dynamic} = C \cdot f \cdot V^2 \tag{3.3}$$

where C is the collective switching capacitance, f is the operating frequency. In the assumed processor model, lowering the supply voltage V restricts the frequency f accordingly because:

$$f \propto (V - V_t)^2/V \tag{3.4}$$

where V_t is the CMOS threshold voltage. The changes in frequency are accompanied by appropriate adjustments in voltage. There is a quadratic relation between the power and the voltage. Thus, lowering the supply voltage can decrease the energy consumption significantly.

3.1.2 Leakage Power Consumption

Leakage power or static power consumption of a CMOS circuit can be simply expressed by (3.5) [26] [33]:

$$P_{leakage} = \sum I_{leakage} \cdot V \tag{3.5}$$

where $I_{leakage}$ is the total leakage current that flow through the CMOS gates of the processor and V is the supply voltage. The dominant types of the leakage power consumption are subthreshold leakage and gate leakage. $P_{leakage}$ is caused mainly by subthreshold leakage current I_{subn} and the reverse bias junction current I_j as the leakage power function defined in (3.6) [26] [33] [37]:

$$P_{leakage} = L_g(VI_{subn} + \mid V_{bs} \mid I_j) \tag{3.6}$$

where L_g is the number of components in the circuit. I_{subn} can be expressed as (3.7) [26] [33] [37]:

$$I_{subn} = k_1 W e^{-V_t/nV_\theta}(1 - e^{-V_/V_\theta}) \tag{3.7}$$

k_1 and n are experimentally derived, W is the gate width, and V_θ in the exponents is the thermal voltage. In order to decrease I_{subn}, we can either lower or turn off the supply voltage V, or increase the threshold voltage V_t. In Formula (3.6), I_j can be expressed as (3.8) [26] [33] [37]:

$$I_j = k_2 W(V_/T_j)e^{-\alpha T_j/V} \tag{3.8}$$

k_2 and α are experimentally derived. T_j is the oxide thickness. Thus, when the cores are idle, only $P_{leakage}$ is consumed, whereas the cores consume both $P_{dynamic}$ and $P_{leakage}$ when executing instructions.

A multi-core processor actually shares some resources, such as caches, interconnection buses, and so on. Our work is based on dynamic repartitioning and dynamic core scaling (Euiseong Seo et al. [44]). We made the same assumptions about these shared components as the previous work [44]. That is, in this work, we do not consider the power consumption from these shared components, and we count only the power consumption of the cores themselves. Our goal is to decrease the energy consumption of the independent cores. Also, we do not consider the energy consumption from sharing these resources and corresponding overhead by using these resources. A CPU could consume less power when it runs in a sleep mode, which is an inactive mode of the CPU when the CPU turns off circuits and signals. There is possible power consumption when the CPU turns on from sleep mode. In this work, we do not consider the overhead caused by the state transition from the sleep mode to the active mode because this overhead can be treated easily in the real-world implementations.

As shown in Figure 3.1 [44], Euiseong Seo et al. simulated the power consumption model of the 70 nm technology based on the original values of Transmetas Crusoe Processor which uses 180 nm technology [33] [37]. The ratio of $P_{leakage}$ to P_{total} increases as f decreases. Above a critical speed (1.2 GHz), the CPU running higher than this speed will consume more $P_{dynamic}$. Thus, P_{total} rapidly increases in the high f domain. Below this speed, $P_{leakage}$ is greater than $P_{dynamic}$.

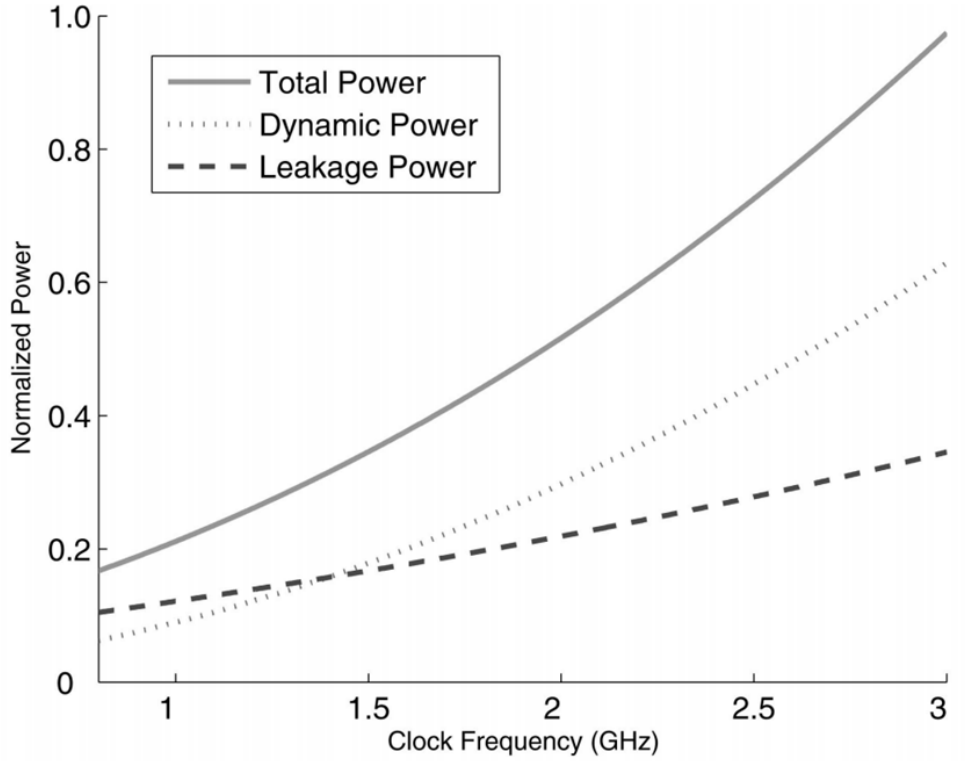


Figure 3.1: Power Consumption of a 70 nm Core as a Function of Clock Frequency [44].

3.2 Task Set Model

As we mentioned in the previous section, the closest work to ours has been done by Euiseong Seo et al. who proposed two energy saving techniques: dynamic repartitioning and dynamic core scaling [44]. Our contributions are threefold. First, we reduce the power consumption of migration overhead in DM. Second, we use the critical frequency to decide the number of the active cores in the system. Third, our algorithm can be applied to multi-core platforms where each core can have different maximum frequencies. As in [44], we also assume a set of n independent periodic real-time tasks as (3.9):

$$T = \{\tau_1, ..., \tau_n\} \tag{3.9}$$

on a set of cores as (3.10):

$$S = \{C_0, ..., C_m\} \tag{3.10}$$

The nth core in S is denoted as C_n. Each core C_i has the variable discrete DVS feature and can adjust its voltage independently of others. For convenience, we normalize the CPU speed with respect to s_{max}, the maximum frequency. We consider the normalized s_i as the following in (3.11):

$$\alpha_i = s_i / s_{max,i} \tag{3.11}$$

Thus, we assume that $s_{max,i}$=1.0. The speed varies between 0 and an upper bound 1.0. s_i represents the relative computational capacity of C_i. α_i represents the scaling factor for the CPU frequency of C_i. The operating frequency is the highest possible frequency of C_i multiplied by the factor α_i if the computational demand on C_i is s_i.

On each core, the voltage is assumed to be dynamically switched upon the release or completion of task instances. A strictly increasing convex function on non-negative real numbers, $g(s)$ can be given to describe the power consumption of the processor under the speed s [13] [14] [24] [45]. The workload of a task is measured by the worst-case number of CPU cycles required for executing the task. If τ_k executes on the cores during the time interval $[T_1, T_2]$ and the speed of the core C_i is given by $s_k(T)$, then the energy consumed during this interval is defined by (3.12):

$$E_k(T_1, T_2) = g(s_k(T))dT \tag{3.12}$$

where $g(s_k(T))$ is calculated before the execution of τ_k and the fixed CPU speed s_k is used during the execution of τ_k.

The period of τ_i is denoted by p_i which is also assumed to be equal to the relative deadline of the current invocation. It means that each instance of a task must complete the execution before starting the next instance of the same task. All the tasks are assumed to be independent and ready simultaneously at T=0. Each partitioned task set is EDF schedulable on its corresponding core. More precisely, we consider a set of n implicit deadlines. Thus, the periodic tasks are denoted as (3.13):

$$T = \{\tau_1(p_1, w_1), ..., \tau_n(p_n, w_n)\} \tag{3.13}$$

where each task τ_i has its own predefined period p_i and WCET w_i. WCET of task τ_i is defined as the maximum execution time required to complete task τ_i at the maximum frequency. Also, each task has its actual execution time which may vary with respect to the system workload. The nearest deadline at the current time is defined as d_i.

The utilization of a task on a core is the ratio of the CPU time required to complete its execution to the period of the task. The utilization u_i of task τ_i is defined by (3.14):

$$u_i = w_i/p_i \tag{3.14}$$

According to [45], the EDF schedule of τ is feasible to be executed on C_i as long as the total utilization of all the tasks in T does not exceed the computation capacity of C_i. U, the total utilization of T, is defined as (3.15) [44]:

$$U = \sum_{\forall \tau_i \in T} u_i \tag{3.15}$$

The system adopts Worst Fit Decreasing (WFD), Best Fit Decreasing (BFD), Next Fit Decreasing (NFD), and First Fit Decreasing (FFD) heuristic partitioned scheduling policies [32] as the initial partition scheduling. After the initial partitioning stages, the partitioned task set allocated to core C_n is denoted as P_n. If each instance of a task τ_i can obtain at least certain CPU cycles to complete before its relative deadline, a schedule of P_n on C_n is feasible. The utilization of P_n is defined by (3.16) [44]:

$$U_n = \sum_{\forall \tau_i \in P_n} u_i \tag{3.16}$$

For simplicity, we further define two functions $\Pi(\tau_i)$ and $\Phi(\tau_i)$ in (3.17) and (3.18) [44]:

$$\Pi(\tau_i) = \{C_j\} \textit{ in which } \tau_i \textit{ was initially partitioned}, \tag{3.17}$$

$$\Phi(\tau_i) = \{C_j\} \textit{ in which } \tau_i \textit{ is currently partitioned}. \tag{3.18}$$

$\Pi(\tau_i)$ indicates the core that τ_i was initially partitioned into. $\Phi(\tau_i)$ indicates the core that τ_i is currently located in.

Based on EDF scheduling, we apply the Cycle-conserving DVS scheduling heuristic [41] on each core. We define the utilization l_i of task τ_i in (3.14) [44] and the utilization L_n of a core running with the Cycle-conserving technique. l_i is updated after every deadline of τ_i.

$$l_i = \begin{cases} w_i/p_i & if\ \tau_i\ is\ unfinished \\ cc_i/p_i & if\ \tau_i\ is\ finished \end{cases} \tag{3.19}$$

l_i is initially equal to u_i as Formula (3.14) is defined. l_i is updated to the most recent actual execution time cc_i replacing w_i in (3.14) after the completion of a task instance. The task will be executed again and the worst case execution time will be used at the beginning of the next period p_i. Thus, the utilization of the task τ_i will be reset to u_i as (3.14) if τ_i is noted as unfinished.

Then L_n, the dynamic utilization of core C_n, is denoted in (3.20) [44]:

$$L_n = \sum_{\forall finished\ \tau_i \in P_n} cc_i/p_i + \sum_{\forall unfinished\ \tau_i \in P_n} w_i/p_i \tag{3.20}$$

L_n is the current computational demand on core C_n. L_n refers to Cycle-conserving utilization of C_n when the context is unambiguous.

3.3 An Energy-efficient Management System Framework

As Figure 3.2 shows, our system performs the following steps: (1) the partitioner assigns all the tasks to each core according to partitioned scheduling algorithms; (2) Local EDF schedulers schedule the tasks in the order of increasing deadlines; (3) a power-aware manager monitors a DVFS controller, a DPM module and a voltage scaling module, and also handles the dynamic values of the accumulated utilization L of all the scheduled tasks; (4) a dynamic migration module uses the updated information from the power-aware manager to manage the task migration among the cores; and (5) a dynamic core scaling module determines if a core needs to be powered on/off.

Lin, Song and Cheng implemented Real-energy [31], a new framework to evaluate the power-aware real-time scheduling algorithms. Six DVS algorithms [31] for unicore systems, including the cycle-conserving EDF scheduler called ccEDF, were implemented and studied in Real-energy. Real-energy is a modulized system. A new DVS unicore algorithm can be easily plugged into the system. Our system extended Real-energy with the new modules for multi-core systems, including a task partitioner, a dynamic migration module and a voltage scaling module. The other advantage to use Real-energy is that six unicore DVS scheduling algorithms were already implemented.

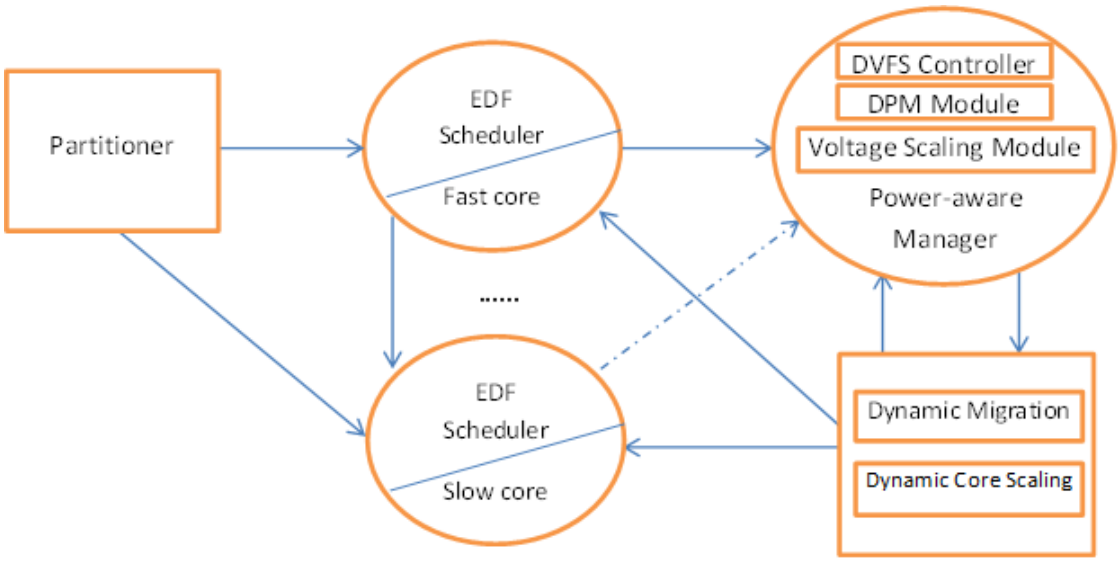


Figure 3.2: System Framework

3.3.1 Partitioner

The partitioner [21] assigns all the tasks to each core according to partitioned scheduling heuristics, e.g., WFD, BFD, FFD, and NFD, as discussed in section 2.1. Thus, we will discuss other components of the system below.

3.3.2 EDF Scheduler

The EDF scheduler [32] has two functionalities. It maintains a task queue for all the tasks and schedules the tasks in the order of increasing deadlines. If a task with an earlier deadline arrives, preemption may be applied to the running task. Each task, τ_i, is released periodically once every p_i (actual units can be seconds or processor cycles) time units. The task needs to complete by its deadline, typically by the end of the period [32]. When a task is released and completed on a local core, the EDF

scheduler [31] selects the task with the nearest deadline among the unfinished tasks on its local core by following the Cycle-conserving EDF algorithm on a schedulable task set and informs the global DVFS controller about this requirement. In order to guarantee meeting the task deadlines, hard real-time scheduling approaches rely on a *priori* knowledge of the worst-case execution times of the tasks. However, in the real world, the task's execution time may be significantly less than its worst-case execution time. It means that the actual computational demands may be lower than the peak computational demands [38]. Thus, when the load is lower, DVFS can dynamically scale the operating frequency to meet the computational requirements and reduce the power consumption.

3.3.3 Power-aware Manager

The power-aware manager in the system controls the voltage of each core in order to reduce the overall energy consumption without violating timing constraints. A global DVFS controller, a DPM module, and a Voltage/Frequency Scaling module are three components of the power-aware manager. The power-aware manager gathers data of the underlying multi-core system during run-time, such as the actual execution time of the tasks, the unfinished tasks, and the current power consumption based on the actual execution time of the tasks, etc.. These values are calculated when a task is released and completed in order to update the current computational demands and the power consumption of all the finished and the unfinished tasks running on the different cores.

3.3.3.1 DVFS Controller

The DVFS controller handles the voltage fluctuations because each task is executed at the supply voltage V_i which can be a fraction of the maximum supply voltage V_{max}. This is because the actual execution time of each task may be less than or equal to its WCET with respect to the system workloads. For the purposes of balancing CPU workloads and lowering the power consumption, the task of re-allocation can be made effective by considering the updates of dynamic utilization and voltage/frequency of all the tasks. The DVFS controller will provide the cores' appropriate voltages/frequencies to fulfill the computational demands received from the EDF schedulers in the system.

3.3.3.2 DPM Module

The DPM module [31] shuts down devices including the processors when they are idle and wakes them up when the tasks have arrived. For example, the PXA270 specification states when the CPU is in a deep-sleep state, the power consumption is only 0.1014mW [8]. When the hardware can support sleep, deep-sleep, and idle states, the DPM can manage the energy consumption by lowering the voltage or powering off the cores [18].

3.3.3.3 Voltage/Frequency Scaling Module

The voltage scaling module selects the voltage/frequency supported by the hardware systems for a core. When a task is released and completed, the power-aware manager

runs the energy-efficient scheduling heuristics based on the updated information, such as the actual execution time of the tasks, the current power consumption, and the current DVFS information. The lower voltage/frequency may be derived to save energy. Moreover, this module will run the voltage scaling algorithm to decide if the current operating voltage/frequency should be changed (Section 4.1 and 4.2). If the operating voltage/frequency should be changed, it will pass the new voltage/frequency value and its core number to the power-aware manager.

The voltage scaling module has been modeled with a series of voltage/frequency levels. For example, the Marvell PXA270 processor supports the dynamic voltage/frequency adjustments. The frequency of the CPU can take one of the following values: 13/104/208/312/416/520/624MHz [3].

3.3.4 Dynamic Migration Module

Balanced distribution of workload among the cores can lower the energy consumption for a multi-core system, which needs online information of the used computational demands on each core to find a solution. As the dynamic frequency is proportional to the power consumption, the related information of the power consumption can be estimated from the actual computational demands. In other words, it is possible to derive a lower speed with task migration when knowing the actual execution times. The task migration is a trigger which tries to reallocate the tasks from a core with higher voltage/frequency to a core with a lower voltage/frequency. In this manner, it can reduce the energy consumption more efficiently. Meanwhile, it needs to guarantee

meeting the deadlines despite abrupt changes in the workload.

To accomplish this, the dynamic migration module is needed to drive the task migration among the cores based on the analysis of the power consumption provided by the power-aware manager. This module maintains the state information of all the tasks. Also, this module checks the voltage scaling module and decides if it is required to migrate the tasks when the voltage scaling module changes frequency for a core. When frequency changes from a higher level to a lower level, the dynamic migration module may start the task migration. After the task migration, the module is updated with new inputs from the completed migrated tasks.

3.3.5 Dynamic Core Scaling Module

The dynamic core scaling module updates the related frequency values provided by the voltage scaling module for each core in the system when a task is released and completed. It runs the VSDCS algorithm and determines which cores are needed to guarantee that the applications can meet their real-time constraints. It will then inform the DPM module to power the cores on/off accordingly.

Chapter 4

Energy-efficient Heuristic Approaches for Multi-core Processors

The energy-efficient management system gains higher energy efficiency by providing the appropriate number of the active cores with dynamic task migration among the cores for multi-core systems. We will discuss two algorithms in this chapter: dynamic migration (DM) and voltage scaling-based dynamic core scaling (VSDCS).

In [44], the authors assume that all the cores must run at the same speed, which is a limitation of their work. Because of the new technologies including Intel Turbo Boost Technology, Intel Enhanced Speed Stepping Technology, and the different C-states [4] [2] [29], multi-core processors have the capability to automatically adjust the core voltage and speed individually. Compared with dynamic repartitioning and

dynamic core scaling, our work can be applied to an asymmetric multi-core system in which the cores can have different clock frequencies.

4.1 Dynamic Migration

Dynamic migration balances task loads between the cores by considering frequently changed computational demands on each core during run-time operation, even though the partitioning heuristic can assign task sets evenly with the well-balanced utilization [16]. Migrating the tasks from a high-load core to a low-load core is an essential way to solve the temporal imbalance and conserve energy. These energy-efficient heuristic algorithms also provide higher energy efficiency by choosing the appropriate number of the active cores for a multi-core system combined with dynamic task migration, which we will discuss in the next section. To explain our algorithms, we introduce the following denotations and analysis.

Let u_i' denote the remaining dynamic utilization of τ_i, which is defined by Formula (3.19) in section 3.2 at a time point T in the current period p_i. Let cm_i denote the executed time [44] of τ_i normalized to the maximum performance by the present time in the period p_i . Let w_i be the worst-case execution time. Let d_i be the next deadline, and T be the present time. Formula (4.1) determines how much processor performance should be reserved for τ_i to be completed from now to d_i [44]:

$$u_i' = \frac{w_i - cm_i}{d_i - T} \tag{4.1}$$

Assuming an instance of a task migrates from the source core C_{src} to the destination core C_{dst} at T, dynamic utilization of C_{src} and C_{dst} defined in (3.19), (i.e. L_{src} and L_{dst}) should be adjusted accordingly. That is u'_i should be reduced from L_{src}. The changed L_{dst} is given by the definition (3.20) in section 3.2. τ_i will be finished before d_i on C_{dst} to meet the deadline constraint. That is, the additional CPU cycles to execute τ_i will be consumed before d_i accordingly. Then, L'_{dst} will be the same as L_{dst} after d_i. Thus, Formula (4.1) will be adjusted to (4.2) [44]:

$$L'_{dst} = \begin{cases} L_{dst} + u'_i & until\ d_i \\ L_{dst} & after\ d_i \end{cases} \tag{4.2}$$

Feasible Scheduling After Task Migration

As Formula (4.2) shows, L'_{dst} is larger than L_{dst} until d_i. A feasible EDF scheduling on the core is ensured before migrating the tasks. That is the maximum L'_{dst} should be guaranteed not to exceed 1.0 before migrating the tasks. Let $M_{dst,di}$ denote the maximum L_n of all the tasks on C_{dst} from the current time to a certain time point T, such that importing τ_i into C_{dst} can be scheduled only when $u'_i + M_{dst,di} < 1.0$ [44].

In this work, the migration of a task is allowed only if it has the nearest deadline among the unfinished tasks on C_{src}. This migration occurs for the current period only. When the next period starts, the migrated task should be returned to its original core assigned by partitioned scheduling policy. However, the task τ_i that was executed on C_{src} can be migrated into C_{dst} again if all the conditions described in the next section have been met when τ_i is released in the next period. A migrated

task may be finished earlier than its worst case execution time, which allows reducing L'_{dst} as (4.3) [44] after combining Formula (4.2) and Formula (3.20) in section 3.2:

$$L''_{dst} = \begin{cases} L_{dst} + \frac{cc_i - cm_i}{d_i - T} & until\ d_i \\ L_{dst} & after\ d_i \end{cases} \tag{4.3}$$

If two conditions, $L_{dst}+u'_i < L_{src} \wedge u'_i + M_{dst,di} < 1.0$ and τ_i has the nearest deadline among the unfinished tasks scheduled on C_{src} [44], are met, the migration operations can be taken recursively. Thus, a task migrated to a core can be exported to the other core within a period as long as all the above conditions are met. Furthermore, if a core has imported an unfinished task, it can export multiple tasks to the other cores [44]. That is, the migration can be overlapped.

4.2 Dynamic Migration Heuristic Algorithm

The DM heuristic algorithm includes Algorithm 1-6 as below.

$\forall \tau_i \in T$, $d_i \leftarrow$ the next deadline of τ_i at every release of τ_i

$\Gamma(C)$ returns a task τ_r such that:

$\forall \tau_i$ where $\Phi(\tau_i) = C$,

$(d_i \geq d_r > 0 \wedge (u'_r > 0) \wedge (\Phi(\tau_r) = C)$

$C_{max} \leftarrow$ core with highest L;

$C_{min} \leftarrow$ core with lowest L;

$M_{n,i} \leftarrow$ the maximum L_n from the calling point to d_i with the current schedule

$A_i = \alpha_{i,1}, ...\alpha_{i,k} \leftarrow$ scaling factors of the i_{th} core in C which has k available frequencies, $\alpha_{i,j} = F_{i,j}/F_{max}$, $\alpha_{i,1} < \alpha_{i,2} < \alpha_{i,k}$

$F_{max} \leftarrow$ maximum frequency of the processors

Algorithm 1 DM Heuristic Algorithm

Input: $T = (\tau_1(p_1, w_1, cc_1), ..., \tau_n(p_n, w_n, cc_n))$
Input: $S = (C_0, C_1, ..., C_m)$
1: on_task_release (τ_i, C_i)
2: on_task_completion (τ_i, C_i)
3: try_to_allocate(τ_i, C_i)

DM Heuristic Algorithm (Our Extension). There are 3 internal operations in this algorithm. The first and the second internal operations are on_task_release() (Line 1 in Algorithm 1) and on_task_completion() (Line 2 in Algorithm 1) defined in dynamic repartitioning algorithm in [44]. The third internal operation try_to_allocate() (Line 3 in Algorithm 1) is called whenever a task is completed or a new task period starts. The function migrates the task in C_{max} with the highest required utilization at that time to C_{min} when the conditions are met. The conditions are: 1) (L_{dst} $+u'_i < L_{src}$) (Line 8 in Algorithm 4); 2) ($u'_i + M_{dst,d_i} < 1.0$) (Line 8 in Algorithm 4); 3) τ_i has the nearest deadline among the unfinished tasks scheduled in C_{src} (Line 5 in Algorithm 4), and 4) the corresponding frequency on the selected destination core for τ_i is close to the higher end of the frequency range, i.e. $(\alpha_{dst,p}+\alpha_{dst,q})/2 < L_i \leq \alpha_{dst,q}$. If τ_i has the nearest deadline among the unfinished tasks scheduled in C_{src}, exporting τ_i can be treated as if it was finished at that time [44]. Condition 4 is designed for two reasons. One is that the available frequencies in the real world are discrete values. Thus, a selection method is needed to choose the frequency value based on the calculated scaling factor. To meet the real-time deadline constraint, a higher or equal discrete frequency value needs to be selected. The other reason is that migration overhead in the real world is non-negligible. Thus, the tasks are not migrated

when the energy savings from the migration is less than the power consumption of the migration overhead. We use a simple heuristic method to decide if a task should be migrated or not: when L_i falls in the higher half range between $\alpha_{dst,p}$ and $\alpha_{dst,q}$, the task τ_i will be migrated; otherwise, no migration occurs. try_to_allocate() function calls try_to_migrate() function (Line 10 in Algorithm 4) to ensure condition 4 is satisfied, which reduces the total number of the task migration compared with [44].

Dynamic Repartitioning Algorithm (Existing Algorithm [44]). There are 3 internal operations in this algorithm. The first and the second internal operations are the same as on_task_release() (Line 1 in Algorithm 1) and on_task_completion() (Line 2 in Algorithm 1). The third internal operation repartitioning() is called whenever a task is completed or a new task period starts. The function migrates the task in C_{max} with the highest required utilization at that time to C_{min} when the following three conditions are met. The conditions are: 1) $(L_{dst} + u'_i < L_{src})$ (Line 8 in Algorithm 4); 2) $(u'_i + M_{dst,d_i} < 1.0)$ (Line 8 in Algorithm 4); 3) τ_i has the nearest deadline among the unfinished tasks scheduled in C_{src} (Line 5 in Algorithm 4).

Algorithm 2 on_task_release [44]

Input: $T = (\tau_1(p_1, w_1, cc_1), ..., \tau_n(p_n, w_n, cc_n))$
Input: $S = (C_0, C_1, ..., C_m)$
Output: $(L_1, ..., L_m)$
1: $\tau_i \leftarrow \Gamma(C)$
2: $\Pi(\tau_i) \leftarrow \Phi(\tau_i)$
3: update L of C(τ_i)
4: try_to_allocate(τ_i, C_i)

Algorithm 3 on_task_completion [44]

Input: $T = (\tau_1(p_1, w_1, cc_1), ..., \tau_n(p_n, w_n, cc_n))$
Input: $S = (C_0, C_1, ..., C_m)$
Output: $(L_1, ..., L_m)$
1: update L of C(τ_i)
2: try_to_allocate(τ_i, C_i)

Algorithm 4 try_to_allocate [44]

Input: τ_i, C_i
Input: $\Phi(\tau_i)$
1: **if** $(\Gamma(C) \neq NULL)$ and $(\Phi(T_i) \neq NULL)$ **then**
2: **while** (true) **do**
3: $C_{src} \leftarrow C_{max}$
4: $C_{dst} \leftarrow C_{min}$
5: $d_i \leftarrow$ nearest deadline of C_{src}
6: u'_i updated by Formula (4.1)
7: L_{src} updated by Formula (3.20) in section 3.2
8: **if** $(L_{dst} + u'_i < L_{src})$ and $(u'_i + M_{dst,d_i} < 1.0)$ **then**
9: **if** $(d_i > d_{\tau_i})$ **then**
10: try_to_migrate(τ_i, C_{src}, C_{dst})
11: **end if**
12: **end if**
13: **end while**
14: **end if**

4.2.1 A Case Study

An example of DM heuristic algorithm with 5 tasks is depicted in Table 4.1. There are 2 cores in the multi-core system. Assume all the tasks are released at $T_1 = 0ms$. The initial partition is decided by the WFD heuristic as shown in the left side of Figure 4.1. The total utilization of C_A is $L_A = 0.2+0.263+0.3 = 0.763$. The total utilization of C_B is $L_B = 0.278 + 0.286 = 0.564$. Thus, the algorithm looks for a task in C_A to migrate. However, no task can be migrated because $L_B + u_i > L_A, i \in (1, 2, 3)$. C_A

Algorithm 5 try_to_migrate

Input: τ_i, C_{src}, C_{dst}
Output: $\Phi(\tau_i)$
1: L'_{dst} updated by Formula (4.2)
2: **if** voltage_scaling(τ_i) = true **then**
3: migrate_task(τ_i, C_{src}, C_{dst})
4: delete_task($\tau_i, \Phi(\tau_i)$)
5: delete_core($C_{src}, \Gamma(C)$)
6: **end if**

Algorithm 6 voltage_scaling

Input: τ_i, C_{src}, C_{dst}
Output: True or False
1: $A_{dst} = (\alpha_{dst,1}, ..., \alpha_{dst,k})$
2: **if** $(\alpha_{dst,p} + \alpha_{dst,q})/2 < L_i < \alpha_{dst,q}, \alpha_{dst,p}, \alpha_{dst,q} \in A_{dst}$ **then**
3: $F_{dst} = F_{max} \cdot \alpha_{dst,q}$
4: return True
5: **else**
6: **if** $L_i = \alpha_{i,1}$ **then**
7: $F_{dst} = \alpha_{i,1} \cdot F_{max}$
8: return True
9: **end if**
10: **end if**

runs at 520MHz ($0.763 \times 624 = 476$MHz) and C_B runs at 416MHz ($0.564 \times 624 = 352$MHz). A task migration happens after the tasks τ_1 and τ_2 are completed at $T_2 = 2$. The total utilization of C_A is updated to $L'_A = 0.2+0.263+0.2 = 0.663$. The total utilization of C_B is updated to $L'_B = 0.278 + 0.143 = 0.421$. All the conditions are satisfied: 1) $L'_B + u'_1 = 0.421 + 0.2 = 0.621 < L'_A = 0.663$; 2) $u'_1 + M_(B, d_2) = 0.2+(0.286+0.278) = 0.764 < 1$; 3) τ_3 has the nearest deadline among τ_3 and τ_5; and 4) the scaling factor to adjust the operating frequency of C_B falls in the higher end of the frequency range, $0.764 > 0.75 = (520MHz/624MHz + 416MHz/624MHz)/2$. The current partition at $T_2 = 2ms$ is depicted in the right side of Figure 4.1.

Task	p	w	cc_1	$\Pi(\tau_i)$	$\Phi(\tau_i)$	u_i	u'_i
τ_1	10ms	3ms	2ms	A	B	0.3	0.2
τ_2	14ms	4ms	2ms	B	B	0.286	0.143
τ_3	15ms	3ms	1ms	A	A	0.2	0.067
τ_4	18ms	5ms	1ms	B	B	0.278	0.056
τ_5	19ms	5ms	3ms	A	A	0.263	0.158

Table 4.1: Example Tasks and System with Core A and Core B

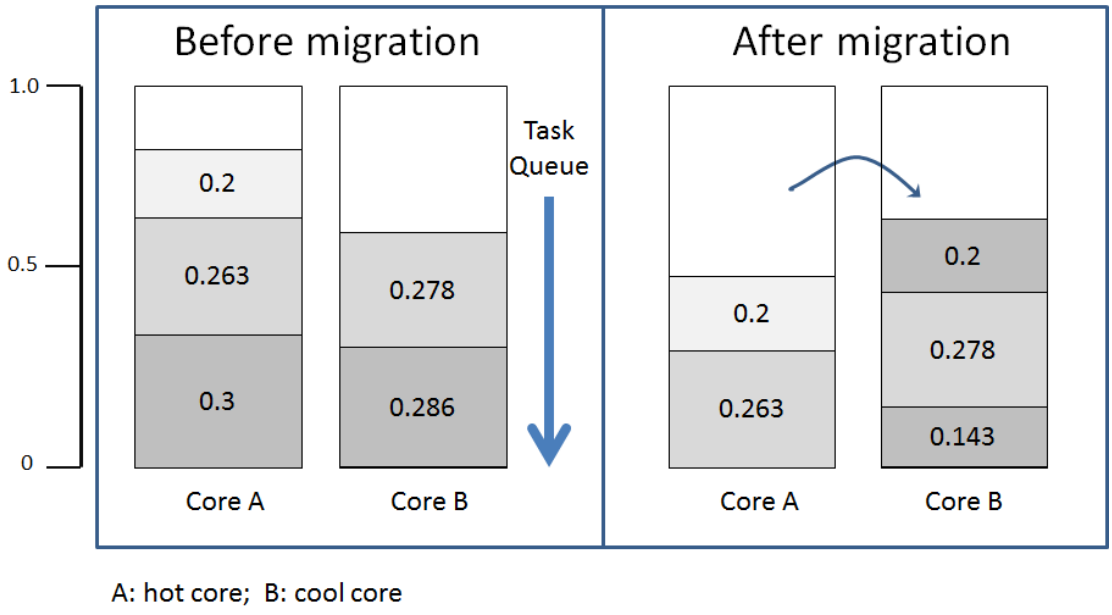


Figure 4.1: An Example of DM Heuristic Algorithm

4.3 Voltage Scaling-based Dynamic Core Scaling Heuristic

Multi-core processors have to consider leakage power increase in order to gain energy efficiency. A fivefold increase in leakage power [17] is predicted with each technology generation. Leakage current is proportional to the total number of circuits as

described in section 3.1. Obviously, a multi-core processor has a greater leakage current than a unicore processor. In [26], the critical speed is defined as the operating point that minimizes the energy consumption per cycle. For example, a critical speed was calculated based on the leakage characteristics of the 70nm technology in [26]. As shown in Figure 3.1, processors using 180 nm manufacturing technology running higher than a critical speed (1.2 GHz) will consume more $P_{dynamic}$. Below this speed, $P_{leakage}$ is the dominant power consumption [44]. Thus, this critical speed is a significant factor in determining the needed number of the active cores for reducing $P_{leakage}$ on multi-core systems. Above this critical speed, more cores can deliver more computational capacity for the real-time applications. Instead, below this critical speed, more cores in fact consume more energy. Thus, our VSDCS is designed based on the critical speed to determine how many the active cores are needed to power off/on dynamically.

Leakage power can be saved at the initial stage by simply adopting the number of the active cores to load the tasks for satisfying computational demands, since most commercial multi-core processors are designed to be able to dynamically adjust the number of the active cores [39] [1]. Since the dynamic utilization, L_n (defined by Formula (3.20) in section 3.2) changes, the frequency of each core is changed accordingly. If the varying frequency is around critical speed, a dynamic core scaling approach can achieve even higher energy efficiency. In the next section, we propose VSDCS algorithm that meets the dynamic requirements of the computational demands while saving energy.

In order to determine the number of the cores by dynamic core scaling algorithm,

Formula (4.4) in [44] is given to define the power consumption expectation function X as below:

$$X(L,n) = n(c_l V_{dd}^2 f + L_g(V_{dd} I_{subn} + |V_{bs}| I_j)) \tag{4.4}$$

and, V_{dd} can be defined by (4.5):

$$V_{dd} = \frac{(f L_d K_6)^{\frac{1}{\theta}} + V_{th1} - K_2 V_{bs}}{K_1 + 1} \tag{4.5}$$

X is affected by the number of the active cores, the supply voltage, the subthreshold leakage, the gate leakage, and the number of components. The subthreshold leakage is determined by the supply voltage. The threshold voltage and the thermal voltage are derived from the subthreshold by Formula (3.7) in section 3.1. Also, the gate leakage is related to the structure of circuits. In the real world, this function is too complex to apply for online applications. Instead, VSDCS considers only the critical frequency $F_{critical}$. Then, we will activate more cores for maximizing the computational demands when $F > F_{critical}$, and try to put more cores in a sleep state when $F < F_{critial}$, where F is the operating frequency of an active core. The following VSDCS algorithm will meet this purpose, as defined in Algorithm 7, 8, and 9 in the next section.

4.4 Voltage Scaling-based Dynamic Core Scaling Heuristic Algorithm

Algorithms:

I is the set of inactivated C

A is the set of actived C

$C_{max} \leftarrow$ core with highest L in A

$C_{min} \leftarrow$ core with lowest L in A

$\Gamma(C)$ returns a task τ_j such that:

$\forall \tau_i$ where $\Phi(\tau_i) = C$, $F_i \leftarrow$ frequency of C

$(d_i \geq d_j > 0 \wedge (u'_j > 0) \wedge (\Phi(\tau_r) = C)$

$F_{critical} \leftarrow$ critical frequency of the processors

Algorithm 7 on_task_completion(t_i)

```
1: update L of C(τ_i)
2: if L · F_{max,i} < F_{critical,i} then
3:    inactivate Φ(τ_i)
4:    try_to_allocate(τ_i, C_i)
5: end if
```

Algorithm 8 try_to_migrate(τ_i, C_m)

```
1: if (voltage_scaling()= True) and (F_dst > F_{critical,dst}) then
2:    if C_dst ∈ I then
3:       I = I - C_dst
4:       A = A + C_dst
5:       activate C_dst
6:    end if
7:    Φ(τ_dst) = migrate_task()
8:    delete_task()
9:    delete_core()
10: end if
```

VSDCS Heuristic Algorithm: It's the same as try_to_allocate() is called when a task releases and completes. However, the critical frequency is checked in try_to_migrate() to ensure that it is energy efficient to activate an inactive core. As described in Algorithm 8, τ_i will be migrated to the selected core if $F_{dst} > F_{critical,dst}$.

The cores with operating frequencies lower than their critical frequencies are inactivated when a task instance completes (Algorithm 8, Line 2-3).

Chapter 5

Evaluation Results and Analysis

5.1 Simulation Environment

STORM (Simulation TOol for Real-time Multiprocessor scheduling) is able to simulate the actual execution time of the real-time tasks and the computational demands during the task execution in terms of the scheduling policies in use, such as EDF and RM on multiprocessor or multi-core systems [43]. The architecture supported by STORM is symmetric multiprocessing (SMP). That is, all the processors in the simulated architecture have the same hardware features. The shared components of SMP architecture, including the memory hierarchy, the shared buses, and the network, are simplified into a defined penalty value. As explained in section 3.1.2, we made the same assumptions as [44] that we do not consider the power consumption of the shared components and we count only the power consumption of the cores. In this case, the abstract SMP architecture is defined by the hardware features of

one processor, and the simulation of the scheduling algorithms on the abstract architecture can reveal the simulated algorithms' behavior on the symmetric multi-core processor. Thus, the core in the simulation is exchangeable with the processor in the experiments.

STORM implements energy conservation methods, such as DPM and DVS [43]. Our tests are aimed to achieve the workload balance and the low power consumption. We integrated DVS and DPM with task migration in order to minimize the power consumption when executing an independent EDF scheduled set of the tasks on one or more cores. STORM can take the task sets and the physical processor description as two inputs. Additionally, preemption is allowed in the STORM environment.

The hardware features of PXA270 are simulated in STORM and we use them for the following simulations. PXA270 is an embedded processor based on the ARM family of Marvell XScale that is supporting DVS. PXA270 [3] can be clocked at six different speeds: 104MHz, 208MHz, 312MHz, 416MHz, 520MHz, and 624MHz. Also, PXA270 supports sleep, deep-sleep, standby, and low power consumption states. When PXA270 is in an idle state, the clock speed is 13MHz. When PXA270 is in a deep-sleep state, there is no power consumed.

Similar with Real-energy [31], STORM allows users to plug in their new scheduling algorithms to run with the underlying abstract architecture. In this study, the algorithms of dynamic migration (DM) and voltage scaling-based dynamic core scaling (VSDCS) are implemented as a new scheduling algorithm. The detailed steps are described in the below.

5.2 Simulation Methodology

As mentioned earlier, some research suggests that worst fit decreasing (WFD) partitioning heuristic technique is the best energy-efficient partitioning approach in comparison with best fit Decreasing (BFD), first fit decreasing (FFD), and next fit decreasing (NFD). We used WFD as the partitioning approach to assign the tasks to each core. To compare with WFD, other partitioning heuristic approaches were used including BFD, FFD, and NFD. Our two algorithms were evaluated by simulating the system model described in Section 3.3. Also, simulation results compared energy consumption of the algorithms we have developed with dynamic repartitioning and dynamic core scaling [44].

DVS/DVFS lowers the voltage/frequency of the multi-core processor, and thus slows the execution of a task. The speed of the multi-core processor is recalculated and updated in terms of the actual execution time after each release and completion of the tasks based on the cycle-conserving DVS algorithm [41].

Based on related research work [41] [11] [16] [15] [23], there are many factors that can affect energy consumption when using the DVS/DVFS policy on multi-core processors. As we mainly focused on energy consumption of the real-time systems, task load, utilization and the actual execution time are four important factors to us. Table 5.1 gives the parameters which can reflect these four factors [44], including utilization, task load, the number of the cores, and the ratio of the execution time to WCET. The generated task sets have a certain upper limit on the utilization u. u follows a normal distribution with $\mu = 0.3$ and $\sigma^2 = 0.2$. We can use the task sets

when u is more than 0.3. However, the number of the tasks running on each core may be less than 3 in order to satisfy the schedulability condition of EDF when u is more than 0.3. α in Table 5.1 is an upper limit on u. The average task load across all the cores is used. It is calculated by dividing U over m.

We randomly generated 1600 task sets with the above desired properties. The tasks were generated for the various combinations of the different numbers of the cores and the different average task load. Currently, dual-core processors are still commonly used in commercial embedded systems and DM was used for embedded multi-core systems. Thus, we also tested our algorithms with 2 cores and the other numbers of the cores tested by dynamic repartitioning algorithm. We generated the task sets for platforms with 2 cores, 4 cores, 8 cores, and 16 cores. The average values of task load U/m are 0.5 and 0.75. Thus, the total number of combinations is 4×2=8, i.e. (m = 2, $U/m = 0.5$), (m = 4, $U/m = 0.5$), (m = 8, $U/m = 0.5$), (m = 16, $U/m = 0.5$), (2)(m = 2, $U/m = 0.75$), (m = 4, $U/m = 0.75$), (m = 8, $U/m = 0.75$), and (m = 16, $U/m = 0.75$). Each task set has at least $(m \times 2 + 1)$ tasks. WCET w and the period of the tasks p are generated randomly following normal distribution with $C++$ template normal_distribution.

Each task set needed to satisfy the following three restrictions. First, the actual execution time has to be between 0.1 and 0.9 times of the worst case execution time, i.e. $cc \in (0.1 \cdot w, 0.9 \cdot w)$ and $em = cc/w$. Second, each task load was less than or equal to 0.3, i.e. $w/p <= 0.3$. Third, the average values of task load are 0.5 and 0.75.

Due to the difficulty to implement dynamic core scaling with 7 hardware variables

Parameters	Values
α	0.3
Number of Cores (m)	4, 8, 16
Task Load (U/m)	0.5 and 0.75
Ratio of cc to WCET (em)	Normal Distribution with μ= (0.3, 0.5, 0.7) and $\sigma^2 = 0.2$

Table 5.1: Parameters Used in the Evaluation [44]

and 8 hardware constants [44], we ran our experiments for DM algorithm, dynamic partitioning [44], and VSDCS.

5.3 Results and Analysis

Figure 5.1 shows a sample simulation result for DM. On the left side of Figure 5.1, the EDF scheduling of several tasks is depicted. Power consumption of 4 cores is on the right side. Power consumption scales from 0mW to 1000mW on the vertical axis and 624MHz corresponds to 1000mW. The horizontal axis shows the elapsed time in millisecond during task execution. Power consumption goes up because the frequency increases. Frequency is dynamically changed based on task load of 35 tasks in this case. We observed that less power is consumed because the lower frequency was used before 150 ms. We also observed that the power consumption increased at several levels, which were around 104MHz, 208MHz, 312MHz, 416MHz, 520MHz, and 624MHz. These frequencies are associated with PXA270 physical features. In general, a discrete set of working frequencies is related to the characters of the processors [41]. There are only several discrete frequency values supported by the underlying hardware.

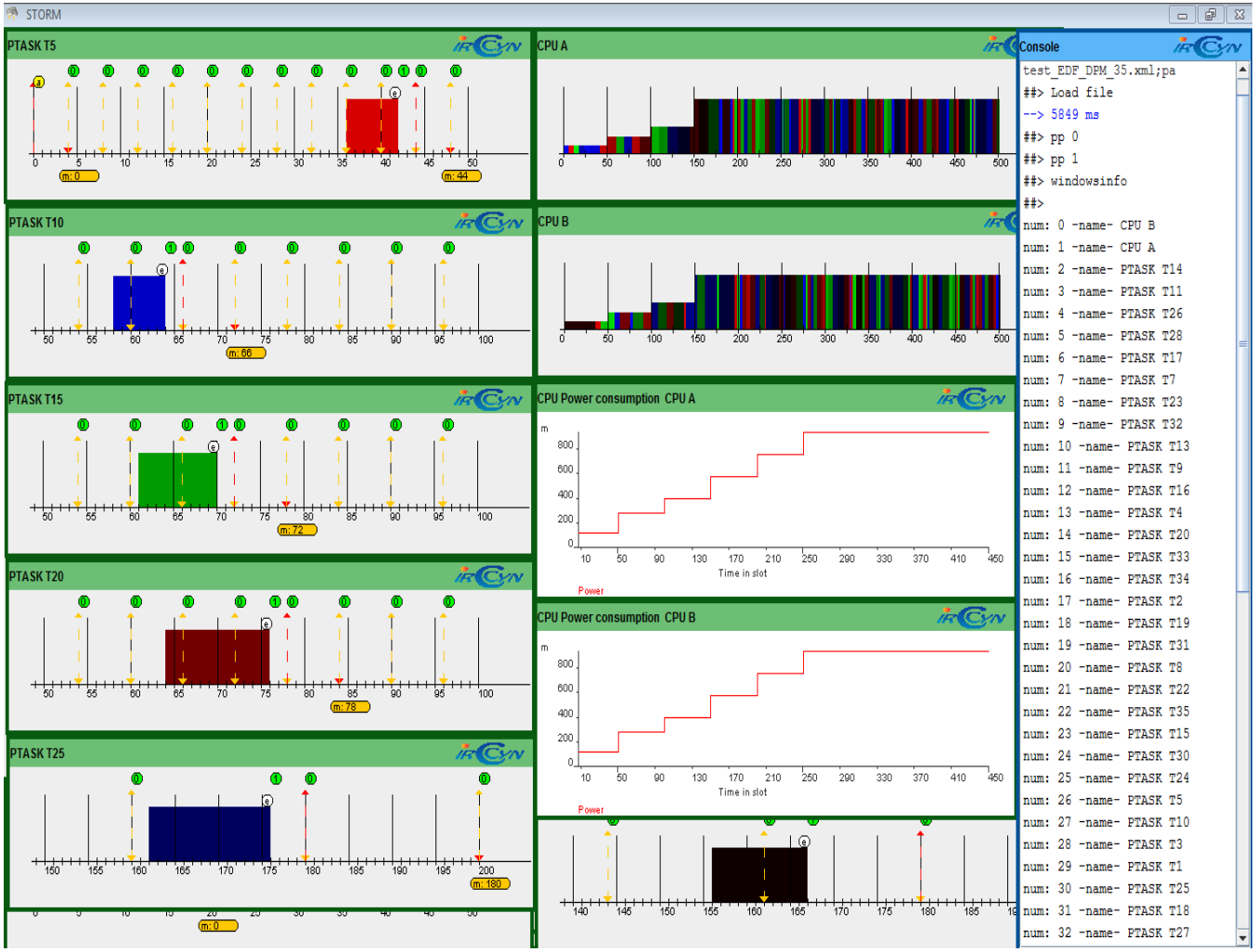


Figure 5.1: 35 Tasks Scheduled on 2 Cores with Average Task Load = 0.5, Average em = 0.6

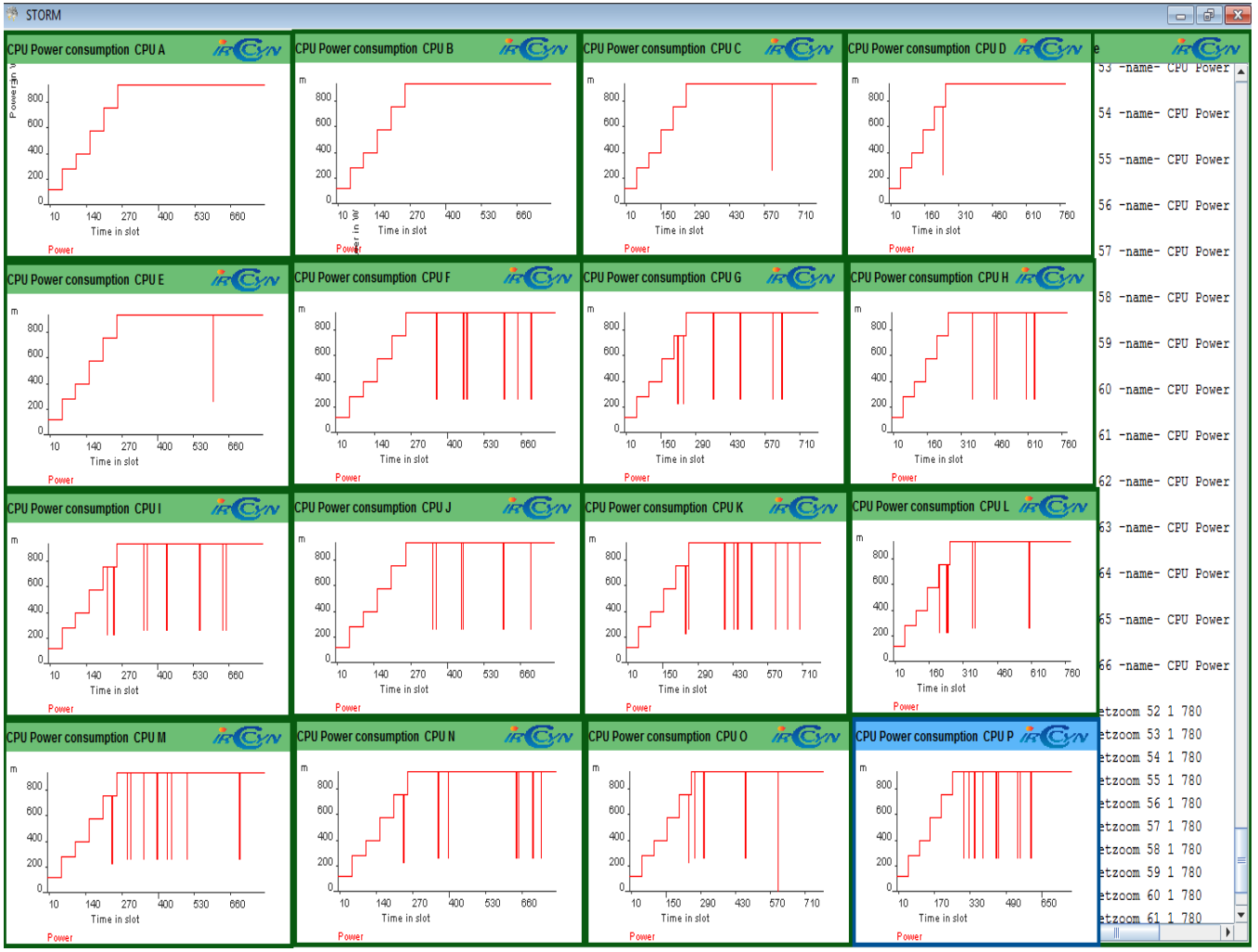


Figure 5.2: 35 Tasks Scheduled on 16 Cores with Average Task Load = 0.5, Average $em = 0.6$

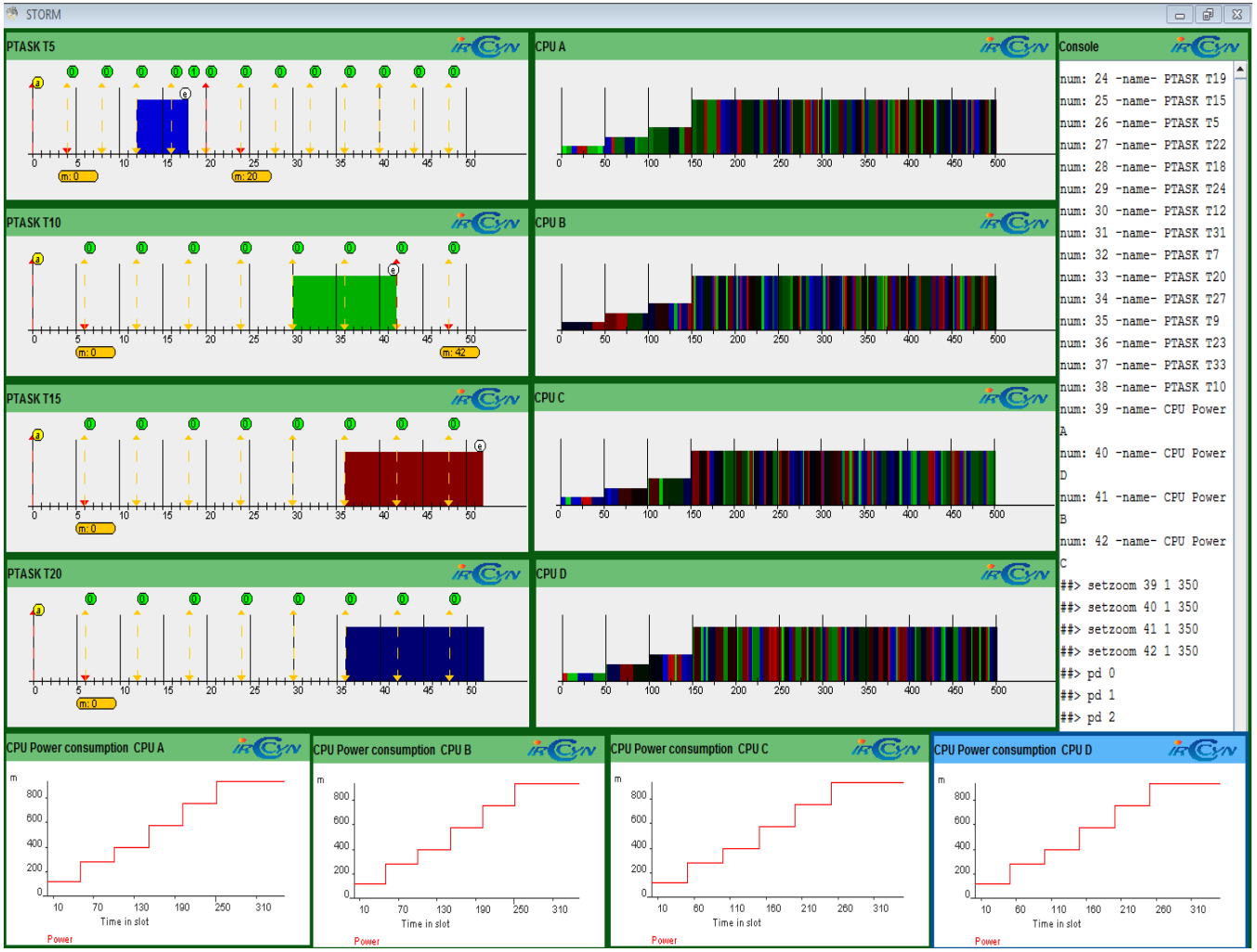


Figure 5.3: 35 Tasks Scheduled on 4 Cores with Average Task Load = 0.5, Average $em = 0.5$

As we mentioned earlier, WFD is the best energy-efficient partitioning approach. Also, WFD is the most energy-efficient partitioning approach compared with BFD, FFD, and NFD when it is used as the partitioning method in dynamic repartitioning [44]. We can observe the obvious difference in energy consumption between BFD and WFD under DM. This is because the task load among the cores is balanced by DM. We primarily compared DM and dynamic repartitioning. The comparison of WFD and BFD is not the purpose of this thesis because DM and dynamic repartitioning both use WFD as the partitioned scheduling heuristic approach.

A number of factors affecting the simulation results are the task set load, the actual execution time, and the number of the cores in the processor.

Number of Cores:

In general, more energy is saved when the number of the cores increases. This holds for both DM and VSDCS. DM algorithm is slightly better than dynamic repartitioning on the task sets run on 2 cores and 4 cores as shown in Figure 5.4(a) and 5.5(a). The difference is about 1 percent energy savings when the task sets run on 2 cores with $em = 0.3$ and the task sets run on 8 cores with $em = 0.3$ as shown in Figure 5.4(a) and Figure 5.6(a). With the same trend, when the task sets run on 16 cores, the energy saving grows as shown in Figure 5.6(a) and Figure 5.7(a). Because the computational demands in each core are higher, more energy is saved.

In Figure 5.4(a) and Figure 5.7(a), when the task load is 0.5 and average em is 0.3, 6 percent more energy is saved when $m = 16$ is compared with $m = 2$ by using VSDCS. This difference is larger when the task load and average em are higher. Since leakage power consumption is reduced by shutting down the cores with

operating frequencies lower than their critical frequencies, VSDCS performs better on a large number of the cores.

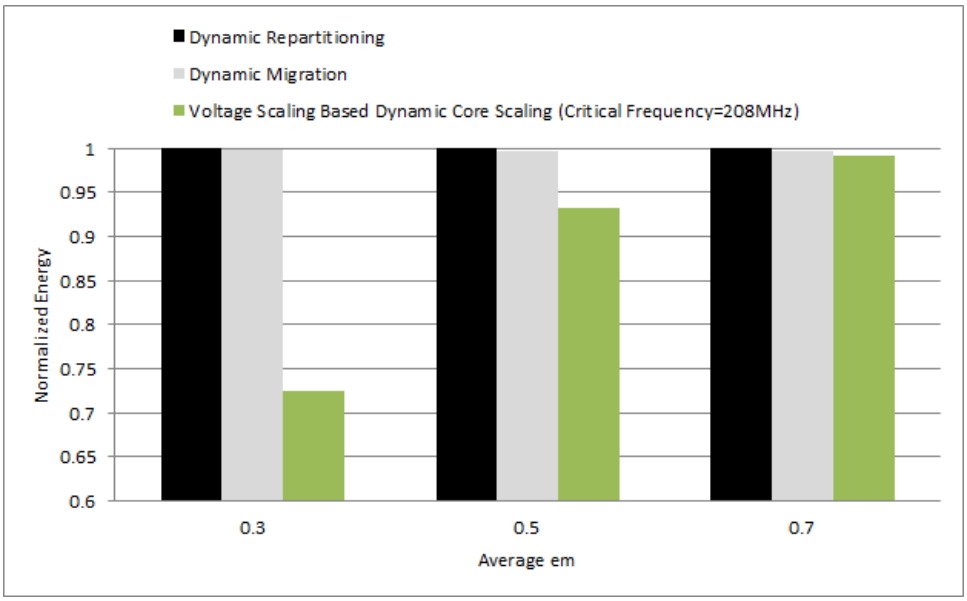
Task Load and Average Actual Execution Time:

To compare DM with dynamic repartitioning, we performed several simulations by varying the task load and the average em for $m = \{2, 4, 8, 16\}$. Figure 5.7(b) shows that DM saves additional 3 percent energy when compared with dynamic repartitioning when $m = 16$, average $em = 0.7$ and $task\ load = 0.75$. This is because when the number of the cores increases, the tasks running on the cores with higher task load have higher possibilities to migrate to the other cores with lower task load. Thus, when $task\ load = 0.75$ and $m = 16$, energy consumption can be reduced up to 3 percent. Regarding DM, the cores have less imbalance of the task load, and thus have lower possibilities to migrate the tasks under low average em as shown in Figure 5.4, 5.5, 5.6, and 5.7. Thus, when em is higher, more energy is conserved. The same holds true for the task load. That is, the higher task load, the more energy savings.

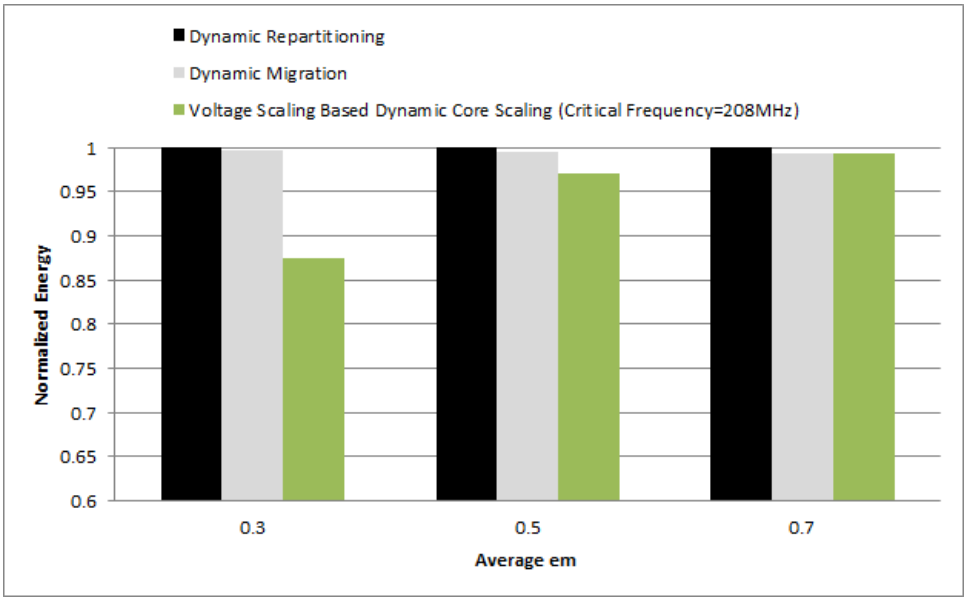
Regarding VSDCS coupled with DM, all the simulation results show that it saves the most energy among three algorithms in comparison. Figure 5.4, 5.5, 5.6, and 5.7 show that more energy is saved when the total task load is lower. Specifically, up to 33 percent of the energy is saved when $taskload = 0.5$ and $em = 0.3$. When VSDCS is applied with DM, up to 22 percent of the energy is saved due to less leakage power consumed when $taskload = 0.5$ and $em = 0.3$.

Experiment setup and process:

STORM is a real time simulator with graphics interface. It currently allows users

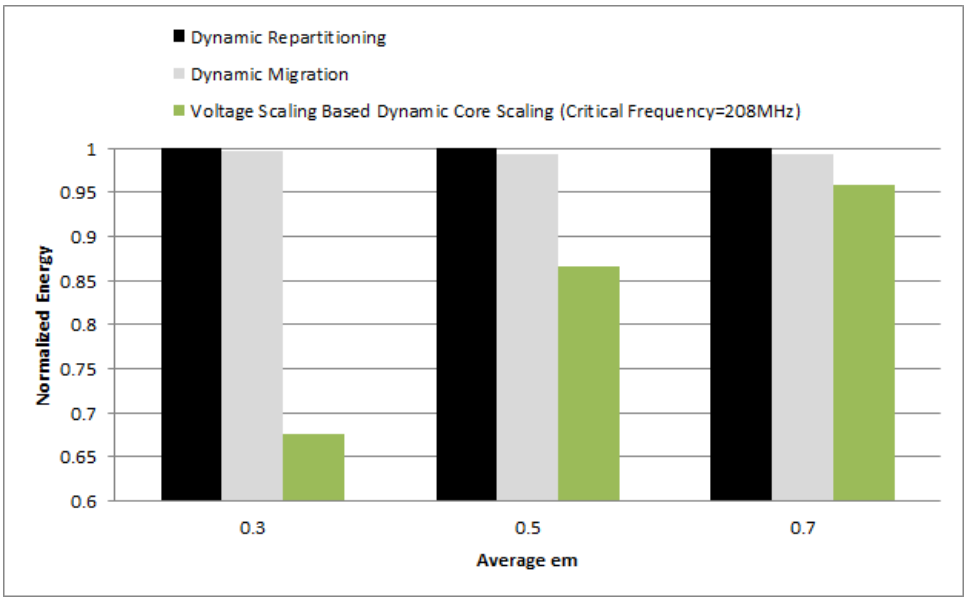


(a) Task load = 0.5; The number of cores = 2; Task sets = 200; The average number of tasks in each task set = 7

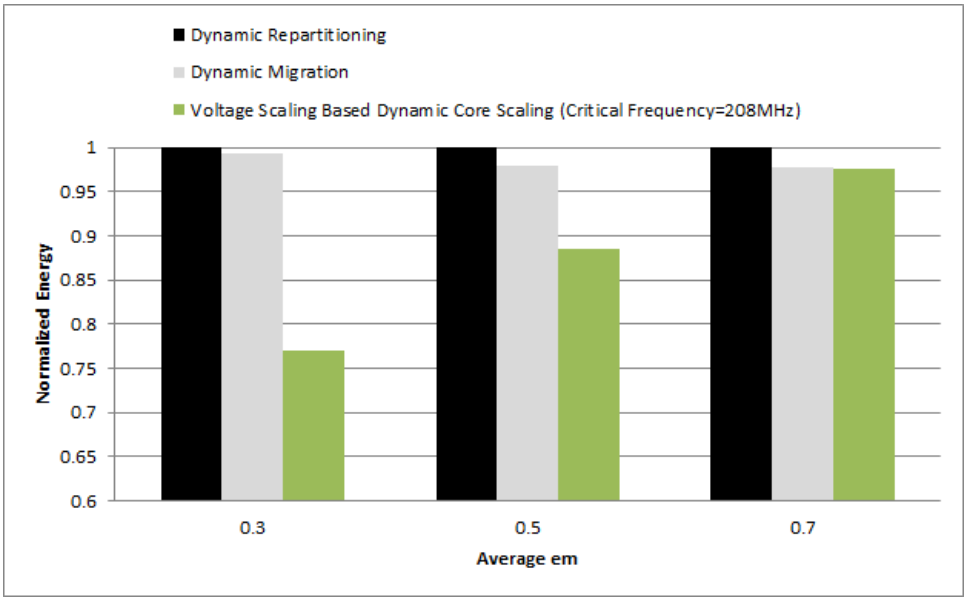


(b) Task load = 0.75; The number of cores = 2; Task sets = 200; The average number of tasks in each task set = 11

Figure 5.4: Normalized Energy Consumption at $m = 2$

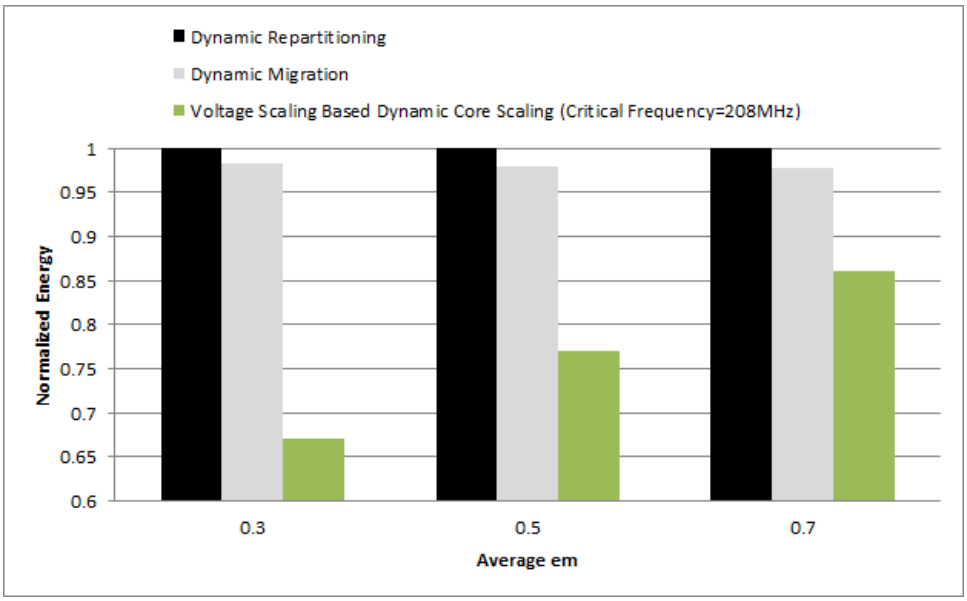


(a) Task load = 0.5; The number of cores = 4; Task sets = 200; The average number of tasks in each task set = 16

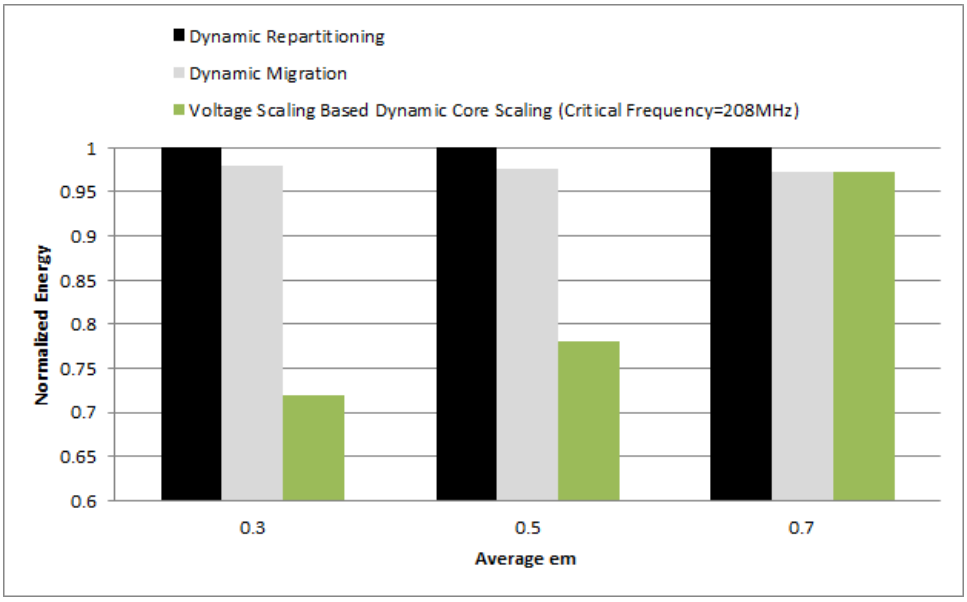


(b) Task load = 0.75; The number of cores = 4; Task sets = 200; The average number of tasks in each task set = 23

Figure 5.5: Normalized Energy Consumption at $m = 4$

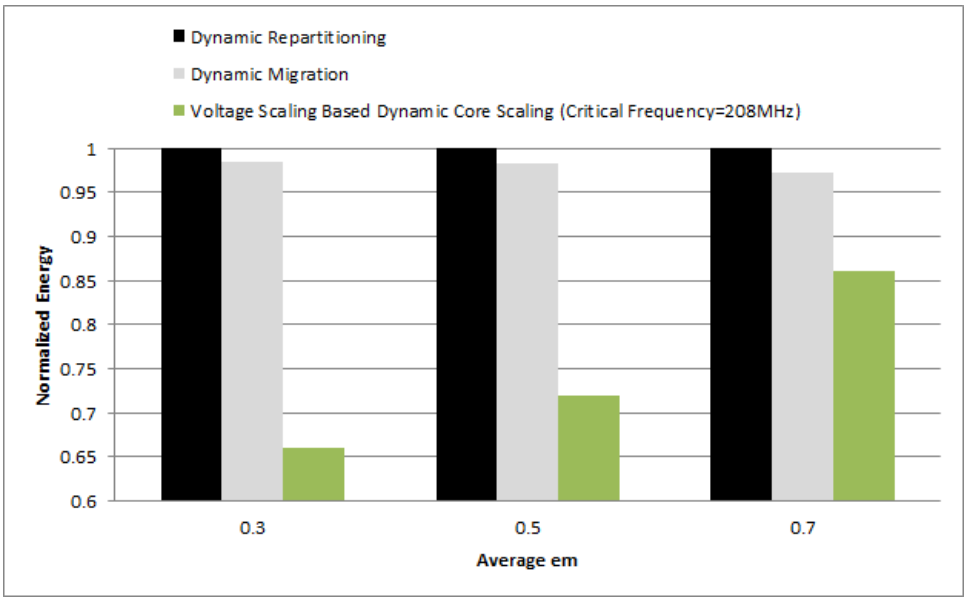


(a) Task load = 0.5; The number of cores = 8; Task sets = 200; The average number of tasks in each task set = 30

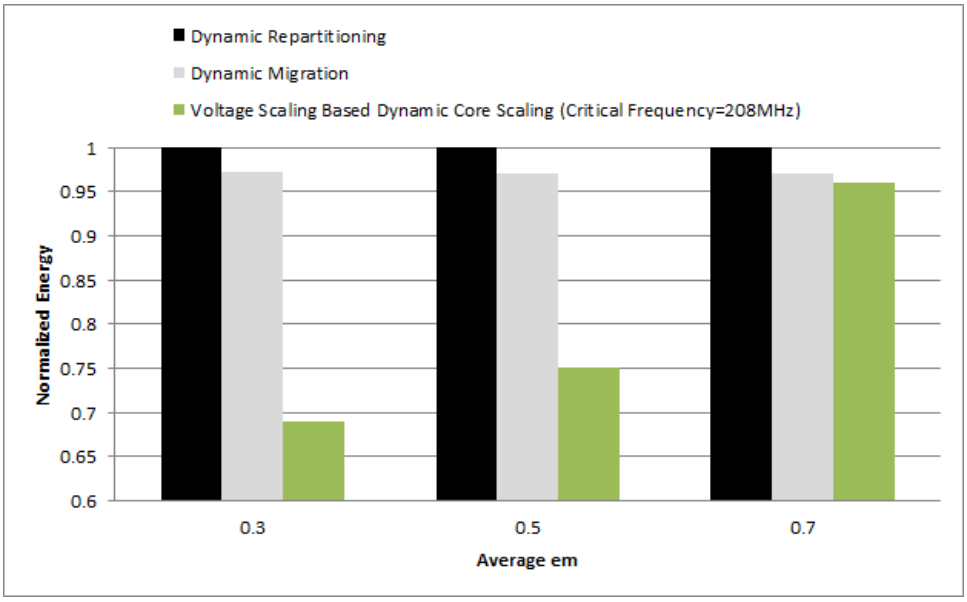


(b) Task load = 0.75; The number of cores = 8; Task sets = 200; The average number of tasks in each task set = 46

Figure 5.6: Normalized Energy Consumption at $m = 8$



(a) Task load = 0.5; The number of cores = 16; Task sets = 200; The average number of tasks in each task set = 120



(b) Task load = 0.75; The number of cores = 16; Task sets = 200; The average number of tasks in each task set = 92

Figure 5.7: Normalized Energy Consumption at $m = 16$

to implement their own real-time schedulers and compile them with the existing framework. It only prints out simulation information within the GUI. The available downloadable package is a jar file without source code available to modify.

To make a new scheduler working with STORM, the implemented scheduler needs to be compiled with the existing framework as the following [9]:

$$javac - cp./storm - 3 - 2\_FL.jarEDF\_P\_FLDCS\_Scheduler.java$$

To use the simulator to schedule a task set, the below steps need to be followed:

1. Launch the simulator with enough memory allocation as the following. STORM now is ready to run scheduling simulation.

   $$java - jar - Xms1024m - Xmx1024mstorm - 3 - 2\_FL.jar$$

   This step is shown in Figure 5.10.

2. Figure 5.8 and 5.9 depict our xml file generator. To schedule a task set, a XML file with the scheduler class name, processor type and information of the tasks in the task set has to be used as the input. This is depicted in Figure 5.11. After the xml file is loaded into the system, the simulation is actually finished. The total time spent on the simulation is printed out. All the information and graphs are now ready to print in the simulator. This is shown in Figure 5.12.

3. There is a set of console commands provided by STORM to print out the graphs and simulator information [43]. To print out the execution time and energy information, the console command $''showdvfs \ < timeslot1 >< timeslot2 >''$ is used as shown in Figure 5.13. Note that STORM can only print DVFS

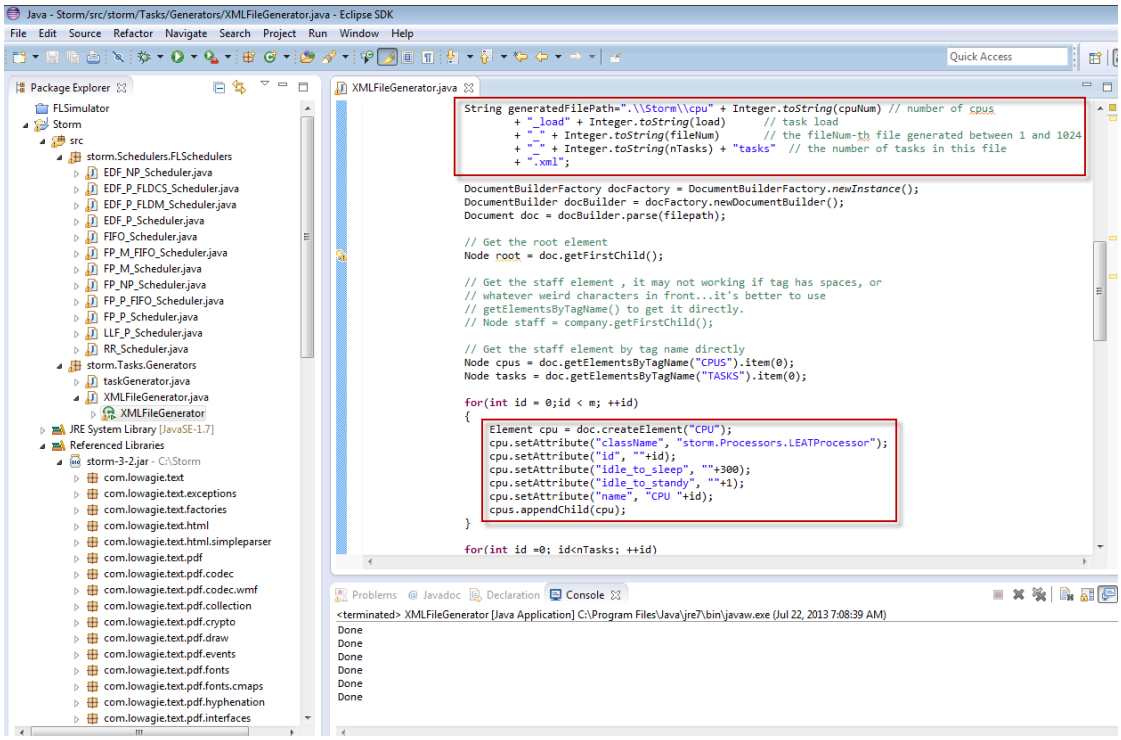


Figure 5.8: XML Generator

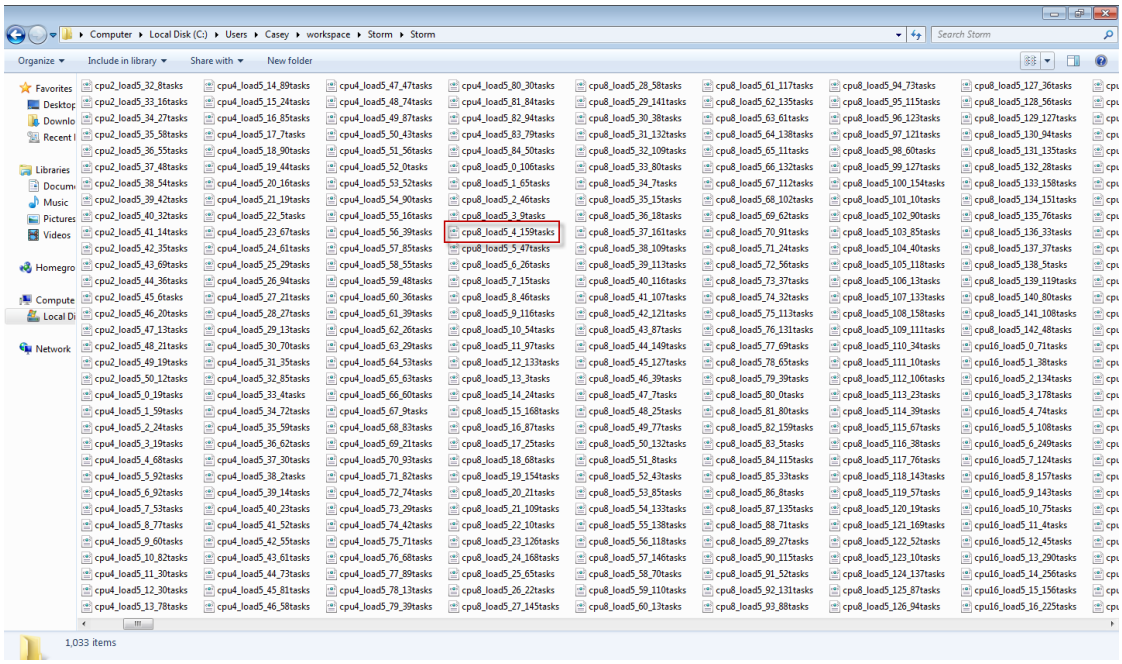


Figure 5.9: XML Files

related information up to time slot 250. The log file of data is also shown in Figure 5.13.

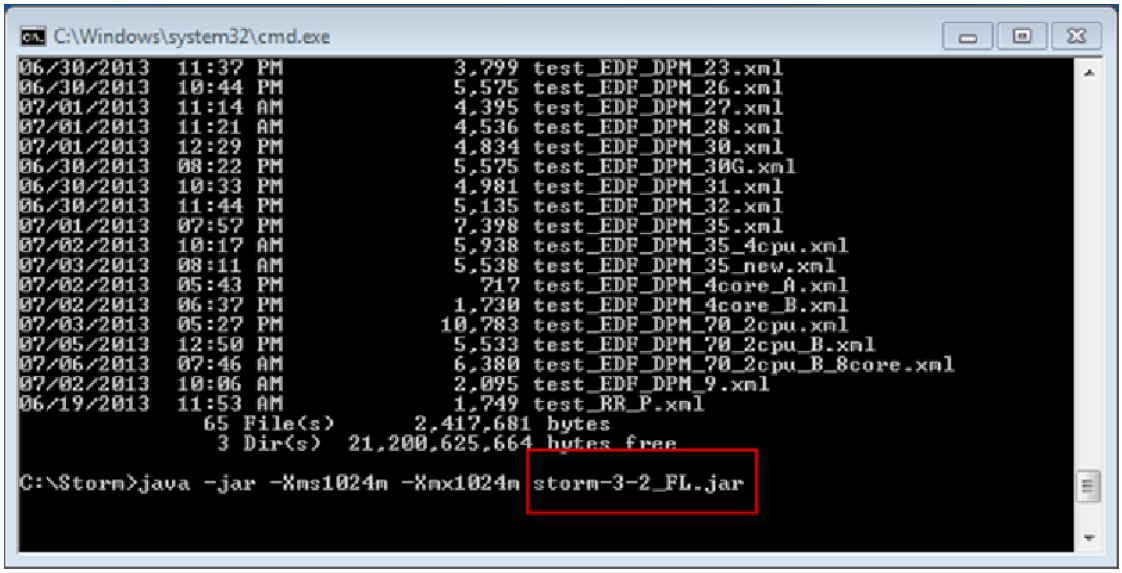


Figure 5.10: STORM Installation

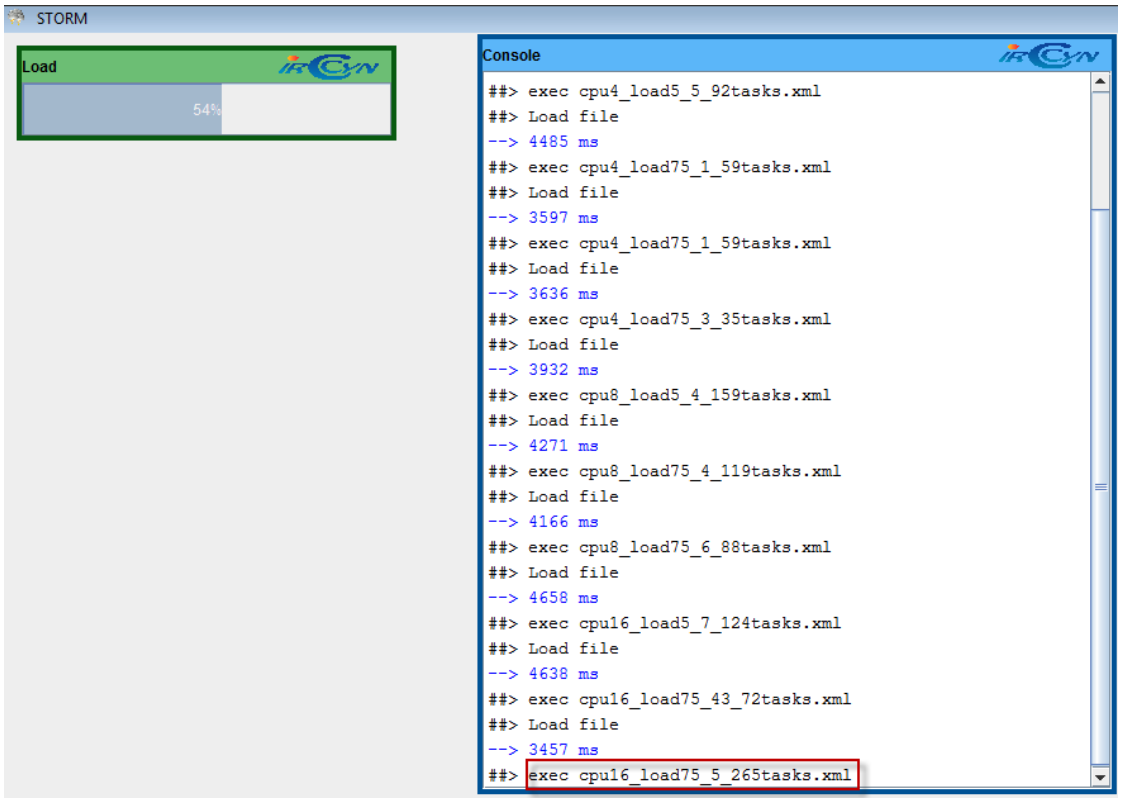


Figure 5.11: Simulation Process-1

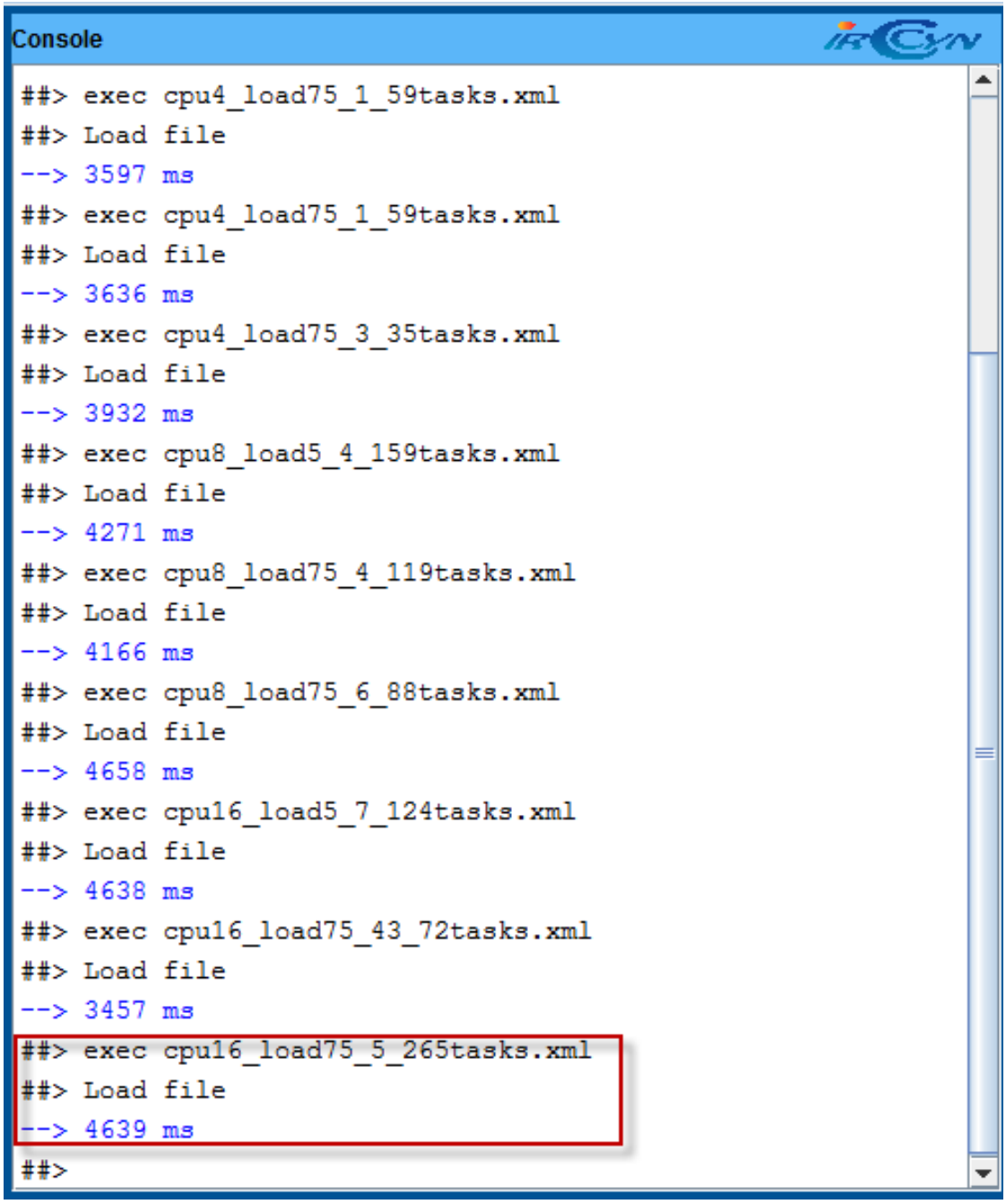


Figure 5.12: Simulation Process-2

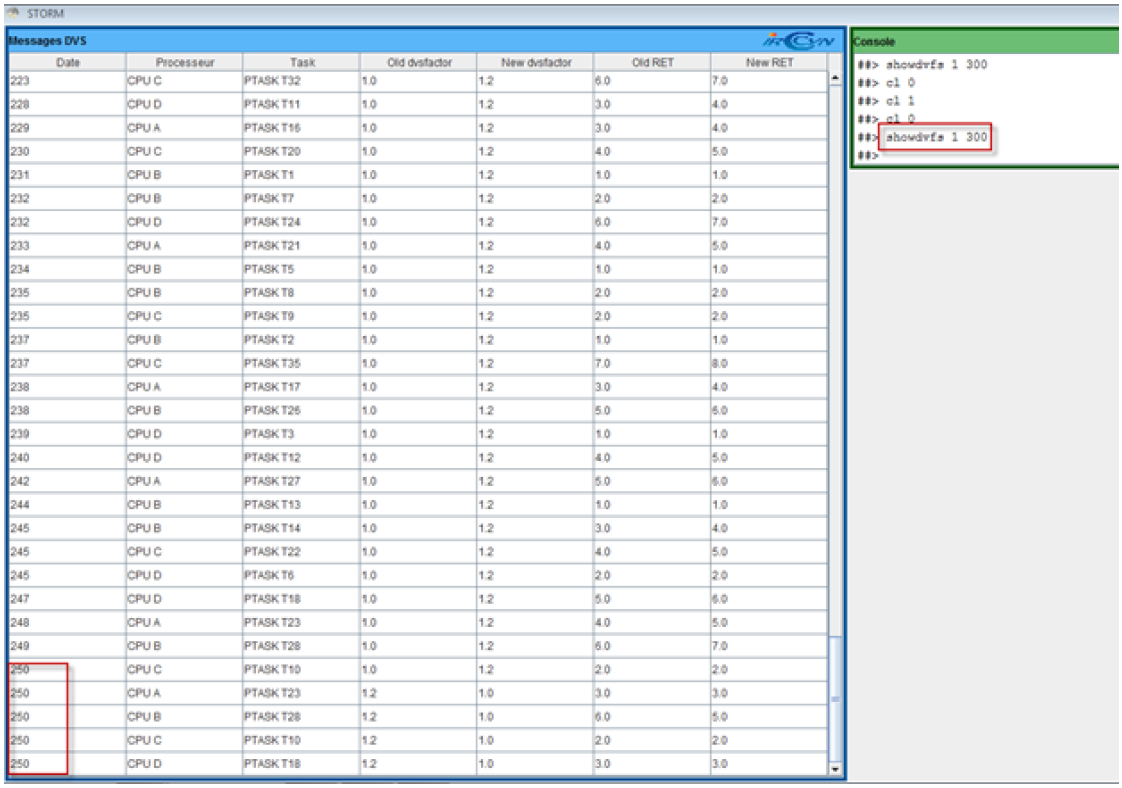


Figure 5.13: Simulation Data

Chapter 6

Conclusions and Future Work

6.1 Conclusions

We introduced an energy-efficient management system model that exploits DVS/DVFS and DPM to save both dynamic and leakage power consumption. This system model includes a partitioner, a local EDF scheduler, a power-aware manager, a dynamic migration module, and a dynamic core scaling module. We extended dynamic repartitioning into dynamic migration (DM) [44]. It migrates the tasks among the cores to balance the computational demands while simultaneously preserving real-time deadline guarantees. DM selects the discrete operating frequency with a simple heuristic to reduce the power consumption of the task migrations. Our voltage scaling-based dynamic core scaling (VSDCS) utilizes the critical frequency to determine the number of the cores instead of calculating the complicated function during run-time execution [44]. Operating at a speed lower than the critical frequency allows the

leakage energy consumption to dominate the overall energy consumption. Also, the leakage power can increase greatly in multi-core processors because of their vastly increased number of integrated circuits in advanced technology generations. Thus, VSDCS considers shutting down the cores when the tasks are executed below the critical speed on the cores.

Simulations predicted several energy consumption characteristics of the system model and two algorithms. Simulation results showed that DM can conserve up to 3 percent energy savings compared to dynamic repartitioning algorithm. It is $(1 - (1 - 8\%) \cdot (1 - 3\%)) = 11\%$ energy saved with the new DM techniques. This is because dynamic repartitioning migrates the tasks among the cores only for balancing the computational demands and ignores the frequency settings of the underlying hardware in the real world. Our algorithm greatly reduces the total number of the task migration. In the related work, dynamic voltage scaling needs to be frequently updated with a complicated power consumption expectation function to determine the number of the active cores. It is very difficult to implement such a function in practice. Our algorithm determines the number of the active cores only by the critical frequency according to the hardware. Thus, our algorithms are simpler to use in the real world. Furthermore, VSDCS can achieve greater energy savings than dynamic repartitioning [44]. Specifically, up to 33 percent of the energy was saved during the simulation experiments.

6.2 Future Work

In the future, we are going to implement this work on the real-world systems. In particular, we will extend Real-energy [31] with our system model and two algorithms on a real-time Linux system to achieve the purpose of this study. We also analyzed the current RT-Linux [5] implementation which supports multi-core systems. According to the requirements of our system design, we decided to develop our algorithms with the combination of RT-Linux and Real-energy.

Bibliography

[1] Advanced configuration and power interface. http://www.acpi.info/spec.htm.

[2] Enhanced intel speedstep technology. http://www.intel.com/cd/channel/reseller/asmona/eng/203838.htm.

[3] Intel pxa270 processor electrical, mechanical, and thermal specification. http://datasheet.eeworld.com.cn/pdf/INTEL/43989-NHPXA270.pdf.

[4] Intel turbo boost technology. http://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html.

[5] Real-time Linux wiki. https://rt.wiki.kernel.org/index.php/Main-Page.

[6] INTEL. strong ARM SA-1100 microprocessor developers manual. 2003.

[7] INTEL-XSCALE. http://developer.intel.com/design/xscale/, 2003.

[8] Marvell PXA270 processor electrical, mechanical, thermal specification. http://www.marvell.com/application-processors/pxa-family/assets/pxa-27x-emts.pdf, 2009.

[9] Simulation tool for real-time multiprocessor scheduling-designer guide v3.3.1. http://storm.rts-software.org/download/Designer-guideV3.3-version1.pdf, 2009.

[10] T. AlEnawy and H. Aydin. Energy-aware task allocation for rate monotonic scheduling. In *Proceedings of the 11th IEEE Real-Time Technology and Applications Symposium (RTAS '05)*, pages 213–223, 2005.

[11] J. H. Anderson and S. K. Baruah. Energy-aware implementation of hard-real-time systems upon multiprocessor platforms. In *Proceedings of the ISCA 16th International Conference on Parallel and Distributed Computing Systems (ISCA PDCS '02)*, pages 430–435, 2002.

[12] B. Andersson, S. Baruah, and J. Jonsson. Static-priority scheduling on multiprocessors. In *Proceedings of the 22th IEEE Real-Time Systems Symposium (RTSS '01)*, pages 193–202, 2001.

[13] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Determining optimal processor speeds for periodic real-time tasks with different power characteristics. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS '13)*, pages 225–232, 2001.

[14] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Dynamic and aggressive scheduling techniques for power-aware real-time systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS '01)*, pages 95–105, 2001.

[15] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Alvarez. Power-aware scheduling for periodic real-time tasks. *IEEE Transactions on Computers*, 53(5):584–600, 2004.

[16] H. Aydin and Q. Yang. Energy-aware partitioning for multiprocessor real-time systems. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS '03)*, pages 9–18, 2003.

[17] S. Borkar. Design challenges of technology scaling. *Micro, IEEE*, 19(4):23–29, 1999.

[18] B. Brock and K. Rajamani. Dynamic power management for embedded systems [soc design]. In *Proceedings of IEEE International SOC Conference (SOCC '03)*, pages 416–419, 2003.

[19] A. Burchard, J. Liebeherr, Y. Oh, and S. H. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Transactions on Computers*, 44(12):1429–1442, 1995.

[20] T. Burd and R. Brodersen. Energy efficient cmos microprocessor design. In *Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences*, pages 288–297, 1995.

[21] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26(1):127–140, 1978.

[22] J. Donald and M. Martonosi. Techniques for multicore thermal management: Classification and new exploration. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA '06)*, pages 78–88, 2006.

[23] F. Gruian. Hard real-time scheduling for low-energy using stochastic data and dvs processors. In *Proceedings of the 17th IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED '11)*, pages 46–51, 2001.

[24] I. Hong, G. Qu, M. Potkonjak, and M. Srivastava. Synthesis techniques for low-power hard real-time systems on variable voltage processors. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS '98)*, pages 178–187, 1998.

[25] R. Jejurikar and R. Gupta. Dynamic slack reclamation with procrastination scheduling in real-time embedded systems. In *Proceedings of the 42th Annual IEEE/ACM Design Automation Conference (DAC '05)*, pages 111–116, 2005.

[26] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *Proceedings of the 41th Annual IEEE/ACM Design Automation Conference (DAC '04)*, pages 275–280, New York, NY, USA, 2004. ACM.

[27] D. S. Johnson. Fast algorithms for bin packing. *Journal of Computer and System Sciences*, 8(3):272–314, 1974.

[28] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd. Power7: IBM's next-generation server processor. *Micro, IEEE*, 30(2):7–15, 2010.

[29] T. Kidd. C-states and P-states are very different. http://software.intel.com/en-us/blogs/2008/03/27/update-c-states-c-states-and-even-more-c-states, 2008.

[30] W. Kim, J. Kim, and S. Min. A dynamic voltage scaling algorithm for dynamic-priority hard real-time systems using slack time analysis. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '02)*, page 788. IEEE Computer Society, 2002.

[31] J. D. Lin, W. Song, and A. M. Cheng. Real-energy: a new framework and a case study to evaluate power-aware real-time scheduling algorithms. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED '10)*, pages 153–158, 2010.

[32] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[33] Y. Liu, H. Yang, R. P. Dick, H. Wang, and L. Shang. Thermal vs energy optimization for dvfs-enabled processors in embedded systems. In *Proceedings of 8th International Symposium on Quality Electronic Design (ISQED '07)*, pages 204–209. IEEE, 2007.

[34] G. Magklis, G. Semeraro, D. Albonesi, S. Dropsho, S. Dwarkadas, and M. Scott. Dynamic frequency and voltage scaling for a multiple-clock-domain microprocessor. *Micro, IEEE*, 23(6):62–68, 2003.

[35] M. Malone. Opera rhbd multi-core. In *Military Aerospace Programmable Logic Device Workshop (MAPLD '09)*, 2009.

[36] J. L. March, J. Sahuquillo, H. Hassan, S. Petit, and J. Duato. A new energy-aware dynamic task set partitioning algorithm for soft and hard embedded real-time systems. *The Computer Journal*, 54(8):1282–1294, 2011.

[37] S. M. Martin, K. Flautner, T. Mudge, and D. Blaauw. Combined dynamic voltage scaling and adaptive body biasing for lower power microprocessors under dynamic workloads. In *Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design (ICCAD '02)*, pages 721–725, 2002.

[38] F. Mueller and J. Wegener. A comparison of static analysis and evolutionary testing for the verification of timing constraints. In *Proceedings of the 4th IEEE Real-Time Technology and Applications Symposium (RTAS '98)*, pages 144–154. IEEE, 1998.

[39] A. Naveh, E. Rotem, A. Mendelson, S. Gochman, R. Chabukswar, K. Krishnan, and A. Kumar. Power and thermal management in the intel core duo processor. *Intel Technology Journal*, 10(2):109–122, 2006.

[40] Y. Oh and S. H. Son. Allocating fixed-priority periodic tasks on multiprocessor systems. *Real-Time Systems*, 9(3):207–239, Nov. 1995.

[41] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 89–102, 2001.

[42] G. Qu. What is the limit of energy saving by dynamic voltage scaling? In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design (ICCAD '01)*, pages 560–563. IEEE Press, 2001.

[43] Y. T. Richard Urunuela, Anne-Marie Dplanche. STORM designer guide 3.3. http://storm.rts-software.org/, 2009.

[44] E. Seo, J. Jeong, S. Park, and J. Lee. Energy efficient scheduling of real-time tasks on multicore processors. *IEEE Transactions on Parallel and Distributed Systems*, 19(11):1540–1552, 2008.

[45] Y. Shin and K. Choi. Power conscious fixed priority scheduling for hard real-time systems. In *Proceedings of the 36th Annual IEEE/ACM Design Automation Conference (DAC '99)*, pages 134–139, 1999.

[46] Y. Shin, K. Choi, and T. Sakurai. Power optimization of real-time embedded systems on variable speed processors. In *Proceedings of the 16th IEEE/ACM International Conference on Computer-aided Design (ISLPED '00)*, pages 365–368. IEEE Press, 2000.

[47] C. Villalpando, A. Johnson, R. Some, J. Oberlin, and S. Goldberg. Investigation of the tilera processor for real time hazard detection and avoidance on the altair lunar lander. In *IEEE Aerospace Conference, 2010*, pages 1–9, 2010.

[48] C.-Y. Yang, J.-J. Chen, and T.-W. Kuo. An approximation algorithm for energy-efficient scheduling on a chip multiprocessor. In *Proceedings of Design, Automation and Test in Europe (DATE '05)*, pages 468–473 Vol. 1, 2005.