Energy Efficient Scheduling of Real-Time Tasks on Multicore Processors

Euiseong Seo, Jinkyu Jeong, Seonyeong Park, and Joonwon Lee

Abstract—Multicore processors deliver a higher throughput at lower power consumption than unicore processors. In the near future, they will thus be widely used in mobile real-time systems. There have been many research on energy-efficient scheduling of real-time tasks using DVS. These approaches must be modified for multicore processors, however, since normally all the cores in a chip must run at the same performance level. Thus, blindly adopting existing DVS algorithms that do not consider the restriction will result in a waste of energy. This article suggests Dynamic Repartitioning algorithm based on existing partitioning approaches of multiprocessor systems. The algorithm dynamically balances the task loads of multiple cores to optimize power consumption during execution. We also suggest Dynamic Core Scaling algorithm, which adjusts the number of active cores to reduce leakage power consumption under low load conditions. Simulation results show that Dynamic Repartitioning can produce energy savings of about 8 percent even with the best energy-efficient partitioning algorithm. The results also show that Dynamic Core Scaling can reduce energy consumption by about 26 percent under low load conditions.

Index Terms—Real-time systems, real-time scheduling, low-power design, power-aware systems, multicore processors, multiprocessor systems.

1 Introduction

Mobile real-time systems have seen rapidly increasing use in sensor networks, satellites, and unmanned vehicles, as well as personal mobile equipment. Thus, the energy efficiency of them is becoming an important issue.

The processor is one of the most important power consumers in any computing system. Considering that state-of-the-art real-time systems are evolving in complexity and scale, the demand for high-performance processors will continue to increase. A processor's performance, however, is directly related to its power consumption. As a result, the processor power consumption is becoming more important issue as their required performance standards increase.

Over the last decade, manufacturers competed to advance the performance of processors by raising the clock frequency. However, the dynamic power consumption $P_{dynamic}$ of a CMOS-based processor, the power required during execution of instructions, is related to its clock frequency f and operating voltage V_{dd} as $P_{dynamic} \propto V_{dd}^2 \cdot f$. And, the relation $V_{dd} \propto f$ also holds in these processors. As a result, the dramatically increased power consumption caused by high clock frequency has stopped the race, and they are now concentrating on other ways to improve performance at relatively low clock frequencies.

- *E. Seo is with the Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA 16803. E-mail: euiseong@gmail.com.*
- *J. Jeong, S. Park, and J. Lee are with the Computer Science Division, Korea Advanced Institute of Science and Technology, 373-1 Guseongdong, Yuseonggu, Daejeon 305-701, Korea. E-mail: {jinkyu, parksy}@calab.kaist.ac.kr, joon@kaist.ac.kr.*

Manuscript received 29 Oct. 2007; accepted 13 June 2008; published online 17 June 2008.
Recommended for acceptance by I. Ahmad, K.W. Cameron, and R. Melhem.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2007-10-0397.
Digital Object Identifier no. 10.1109/TPDS.2008.104.

One of the representative results from this effort is multicore architecture [1], which integrates several processing units (known as cores) into a single chip. Multicore processors, which are quickly becoming mainstream, can achieve higher throughput with the same clock frequency. Thus, power consumption in them is a linear function of the throughput. As the demand for concurrent processing and increased energy efficiency grows, it is expected that multicore processors will become widely used in real-time systems.

The problem of scheduling real-time tasks on a multicore processor is the same as that of scheduling on a multiprocessor system. This is an NP-hard problem [2], and existing heuristic solutions can be divided into two categories. Partitioned scheduling algorithms [3], [4], [5] require every execution of a particular task to take place in the same processor, while global scheduling algorithms [6], [7], [8] permit a given task to be executed upon different processors [6]. Partitioned algorithms are based on a divide-and-conquer strategy. After all tasks have been assigned to their respective cores, the tasks in each core can be scheduled using well-known algorithms such as *Earliest Deadline First* (EDF) [9] or *Rate Monotonic* (RM) [10]. Due to their simplicity and efficiency, partitioned scheduling algorithms are generally preferred over global scheduling algorithms.

In addition to the innovation of multicore architecture, many up-to-date processors also use dynamic voltage scaling (DVS). DVS adjusts the clock frequency and operating voltage on the fly to meet changes in the performance demand.

Multicore processors can also benefit greatly from DVS technology. Because all the cores in a chip are in the same clock domain, however, they must all operate at the same clock frequency and operating voltage [11], [12]. It seems

that this limitation will remain in force for some years at least because the design and production of multicore processors with independent clock domains is still prohibitively expensive.

There has been much research [13], [14], [15], [16], [17] on how best to use DVS in a unicore processor for real-time tasks. In systems consisting of multiple DVS processors, DVS scheduling is easily accomplished using those existing algorithms on each processor after partitioning [13], [14], [15]. In multicore environments, however, the benefit of this approach is greatly reduced by the limitation that all cores must share the same clock. Even though the performance demands of each core may differ at a given scheduling point, this limitation forces all cores to work at the highest frequency scheduled. Compared to a multiprocessor system, a multicore system will thus consume more power needlessly if the existing DVS method is adopted blindly.

This paper suggests a dynamic, power-conscious, real-time scheduling algorithm to resolve this problem. In general, multicore processors have some caches that are shared among their cores. Task migration between cores thus requires less overhead than migration between fully independent processors. With an exploitation of this property, *Dynamic Repartitioning*, which is the suggested scheme tries to keep the performance demands of each core balanced by migrating tasks from the core with the highest demand to the one with the lowest demand. Similar to multiprocessor systems, the dynamic performance demand of each core is given by existing DVS algorithms, and the migration decisions are made at every scheduling time. This repartitioning of tasks is expected to reduce the dynamic power consumption by lowering the maximum performance demand of the cores at any given moment.

In addition to dynamic power, there is another source of power consumption that must be considered. Different from dynamic power, which is consumed during instruction execution, leakage power is consumed as long as there is electric current in the circuits. In general, this energy loss is proportional to the circuit density and the total number of circuits in the processor. Leakage power has thus been taking up an increasing proportion of the total power, up to 44 percent in 50 nm technology for an active cycle of a uniprocessor [18]. And, it will become even more in a multicore processor for the vastly increased circuits.

In this paper, we also suggest a method of reducing the leakage power by adjusting the number of active cores. *Dynamic Core Scaling* decides on the optimal number of cores for the current performance demand and tries to meet this criterion as far as all deadlines are guaranteed. Dynamic Core Scaling is expected to save a considerable amount of leakage power in low load periods, where the leakage power makes up a large fraction of the total power consumption.

The suggested Dynamic Repartitioning and Dynamic Core Scaling methods were evaluated through simulations by applying them to a well-known processor power model. The target task sets in the simulations were designed to demonstrate the behavior of the algorithm under diverse environments.

TABLE 1
Example Task Set [13]

Task	Period	WCET	Utilization	cc_1	cc_2
τ_1	8 ms	3 ms	0.375	2	1
τ_2	10 ms	3 ms	0.300	1	1
τ_3	14 ms	1 ms	0.071	1	1

The rest of this paper is organized as follows: Section 2 reviews existing research on the use of DVS in real-time unicore processor systems and on the development of energy-efficient scheduling algorithms in multiprocessor and multicore systems. Section 3 defines the problem and describes the power consumption model used in this paper. In Section 4, we describe Dynamic Repartitioning algorithm as a way of efficiently reducing clock frequencies. In Section 5, we introduce Dynamic Core Scaling algorithm, which reduces the leakage power by adjusting the number of activated cores. Section 6 presents simulation results for the two algorithms, and Section 7 summarizes our conclusions.

2 Related Work

2.1 DVS on a Unicore Processor

In this paper, the WCET of a task will be taken as the time required to finish the worst-case execution path at maximum performance. The actual WCET of a task is the scaled value of its WCET to the current performance, and it increases linearly as performance degrades. In this paper, we will use the term *utilization* of a task to refer to its WCET divided by its period. It means the fraction of processor time dedicated to the task at maximum performance. A matter of course, the relative utilization of a task which is based on its actual WCET is also grows as the performance degrades.

EDF is the optimal algorithm for preemptible periodic real-time task scheduling. Defining the *utilization* of a task as the value of its WCET divided by its period, EDF can guarantee meeting the deadlines of all task sets for which the sum of all task utilization is less than one. Based on this property, Pillai and Shin [13] suggested three DVS scheduling heuristics: Static, Cycle conserving, and Look ahead.

The Static algorithm adjusts the clock frequency so that the total relative utilization of all tasks is 1.0. For example, the total relative utilization of the task set in Table 1 is 0.746. If the execution time of all tasks is inversely proportional to the clock frequency, then we can achieve the highest possible energy efficiency while still meeting all deadlines by scaling the performance down to 0.746 times the maximum clock frequency.

A given task may be finished earlier than its WCET, and the actual execution time changes with every period. If the tasks in Table 1 have actual execution times as given in columns cc_1 and cc_2 during their first and second periods, respectively, then idle periods will result even executing at the frequency given by the Static algorithm. This means that the frequency could be lowered further.

To exploit this phenomenon, the Cycle-conserving algorithm adopts the notion of dynamic utilization, which

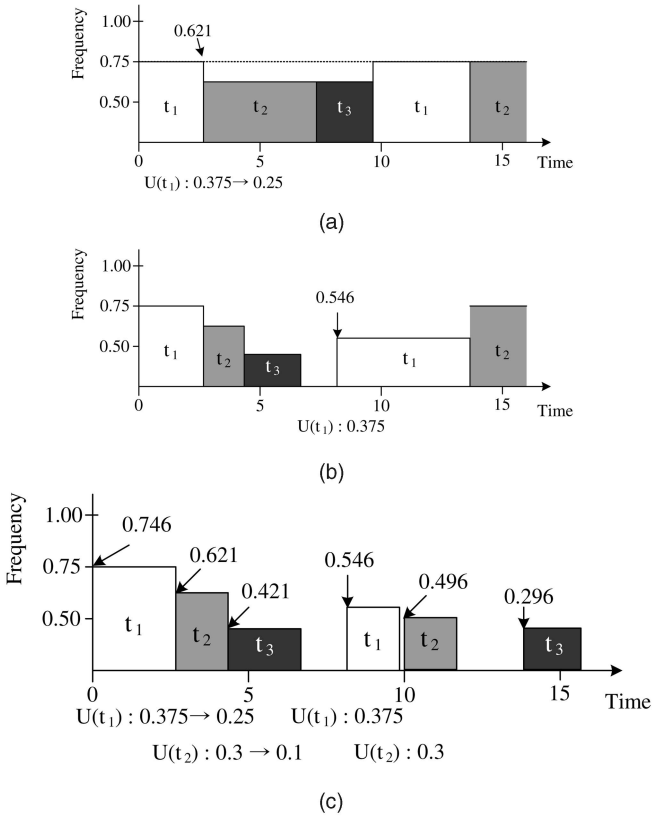

Fig. 1. Cycle-conserving algorithm on the example task set [13]. (a) After finish of executing τ_1. (b) After finish of executing τ_2 and τ_3. (c) Actual execution flow for two rounds.

is updated whenever the tasks are scheduled and finished. On the completion of a task, it updates the utilization based on the task's actual execution time. The next time the task begins executing, however, its utilization is restored to the original value based on the task's WCET. In this manner, the Cycle-conserving algorithm may choose a lower frequency than the Static algorithm during the period between a task's completion and the start of its next period. It thus saves more energy than the Static algorithm.

Fig. 1 shows an example of the Cycle-conserving algorithm at work. The actual execution time of τ_1 is 2 ms (Fig. 1a). The utilization of τ_1 is thus updated from 3/8 to 2/8 after its first execution, and the total utilization of the task set decreases to 0.621. At this point (Fig. 1b), the processor will be operated at 0.621 times the highest frequency. The utilization of τ_2 drops from 3/10 to 1/10 after completion, and as a result, τ_3 can be executed at 0.421 times the highest frequency. The actual execution flow under the Cycle-conserving algorithm for both rounds is shown in Fig. 1c.

Cycle conserving is expected to lead to a higher energy efficiency than the Static algorithm, because it reduces the frequency during idle periods. As shown in Fig. 1c, at the start of each new period, it assumes the worst case for the current task. As a result, the frequency tends to start high and decrease gradually as tasks are completed. If the actual execution times of most tasks fall short of their WCET, however, it is better to start with a low frequency and defer the use of high-frequency processing as long as all deadlines can be met. This is the basic concept of the Look-ahead algorithm. When actual execution times usually fall short of their corresponding WCETs, the Look-ahead algorithm gives better results than Cycle-conserving. In cases where the actual execution times are usually close to their WCETs, however, Cycle conserving is the better choice.

A variety of DVS algorithms have been proposed in addition to these. Aydin et al. [14], for example, have suggested the Generic Dynamic Reclaiming Algorithm (GDRA) and Aggressive Speed Adjustment (AGR) algorithms. GDRA is in many respects similar to the Cycle-conserving algorithm; AGR, however, sets the frequency based on the execution history of the tasks. Gruian [15] suggested an algorithm that starts at a low frequency and increases the processing speed gradually based on the statistics of the actual execution times. Kim et al. [17] also suggested the method to utilize slack time, which is based on the expectation of the slack time occurrences. These alternative approaches are helpful in cases, where trends are visible in the actual execution times, for example, when the most recent execution time is related to the previous one.

2.2 Power-Aware Scheduling on Multiprocessors

Besides the problem of deciding which task to execute at a certain time, multiprocessor real-time systems must also decide which processor the task will run on. Partitioned scheduling is the most widely used solution to this NP-hard problem. In partitioned scheduling, every processor has its own task queue, and in an initial stage, each task is partitioned into one of these queues. Each processor's task set is scheduled with a single-processor scheduling algorithm such as EDF or RM [3], [5]. The partitioning itself is one variant of the well-known Knapsack problem, for which a number of heuristics such as Best Fit, Worst Fit, and Next Fit are known to work well.

The partitioning approach has the advantage of utilizing DVS. Any of the many possible DVS algorithms described in Section 2.1 can be used to adjust the frequency of each processor and its associated task set. To maximize the energy efficiency, however, the utilizations of each partitioned set should be well balanced [4]; this is because the dynamic power consumption increases as the cube of the utilization.

Aydin and Yang [4] proved that it is also an NP-hard problem to partition a list of tasks into a given number of sets that are optimally load balanced, with the guarantee that each task set can be scheduled on the system. They also showed that among well-known heuristics, worst fit decreasing (WFD) generates the most balanced sets. WFD applies the worst-fit algorithm to the tasks after sorting them in order of decreasing task utilization.

There are also many scheduling heuristics for the variety configurations of target environments. Gruian [19] proposed a simulated annealing approach in multiprocessor energy efficient scheduling with the considerations of precedence constraints and predictable execution time for each task. Chen et al. [20] suggested an approximation algorithm with different approximation bounds for processors with/without constraints on processor speeds for the task set with common periods. Anderson and Baruah [21] suggested the

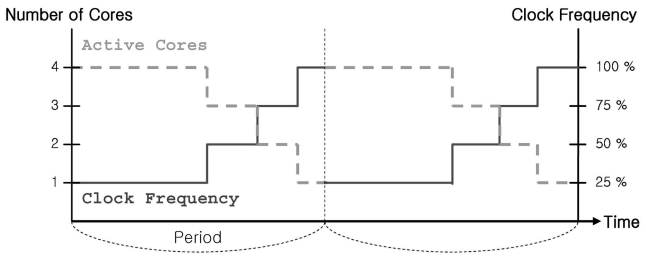

Fig. 2. Example schedule generated by heuristic algorithm [12] for DVS-CMP.

trade-off between increasing the number of processor and increasing the performance of each processor is explored, and they also suggested algorithms to solve the problem with static analysis. However, even though the many works have been done, most of them are based on the static analysis of the WCETs of tasks and have little consideration for utilizing slack time.

2.3 Power-Aware Scheduling on Multicores

While much research has examined the problem of energy-efficient scheduling for single-processor or multiprocessor systems, little work has been done on multicore processors.

Nikitovic and Brorsson [22] assumed an adaptive chip-multicore processor (ACMP) architecture, in which the cores have several operating modes (RUN, STANDBY, and DORMANT) and can change their state dynamically and independently according to the performance demand. They suggested some scheduling heuristics for ACMP systems and demonstrated that these heuristics save a significant amount of energy for non-real-time task sets compared to a high-frequency unicore processor. Although this work introduced the benefits of processors with multiple cores that can change their operating mode independently, it does not take into consideration the demands of real-time task sets.

The first energy-efficient approach to real-time scheduling on a multicore processor was suggested by Yang et al. [12], who assumed a DVS-enabled chip multiprocessor (DVS-CMP). In DVS-CMP systems, all cores share the same clock frequency but a core can "sleep" independently if it has no work to do. Yang et al. proved that the energy efficient scheduling of periodic real-time tasks on DVS-CMP system is an NP-hard problem. They thus suggested a heuristic algorithm for scheduling a framed, periodic, real-time task model. In this model all tasks have the same period, share a deadline which is equal to the end of the period, and start at the same time. As shown in Fig. 2, the suggested algorithm starts executing tasks at a low performance. As time goes on, cores with no tasks to run will be set to the sleep state. When the number of sleeping cores increases, the frequency must also increase to meet the deadlines of tasks that have not been finished yet. In this manner the number of cores running in a high frequency mode is reduced, and a significant amount of energy will be saved. The applications of this algorithm are limited, however, because it can be only used for the framed real-time systems in which all tasks have same dead-lines and starting points. Moreover it is also a static approach. In other words, it does not take into account cases where the actual execution times may be shorter than the WCETs, which are close to the real world. If this is so, then additional energy can be saved with a dynamic approach.

3 System Model

3.1 Task Set Model

The assumed target tasks are executed periodically, and each should be completed before its given deadline. A completed task rests in sleep state until its next period begins, at the start of which the task will again be activated and prepared for execution. The tasks have no interdependency.

A task set $\mathcal{T}$ is defined by (1), where τ_i is the ith individual task in $\mathcal{T}$. Each task has its own predefined period p_i and WCET w_i; the latter is defined as the maximum execution time required to complete τ_i at the highest possible processor frequency. The real worst-case execution time of τ_i thus increases from w_i as the clock frequency decreases. The nearest deadline at the current time is defined as d_i:

$$\mathcal{T} = \{\tau_1(p_1, w_1),\ \ldots,\ \tau_n(p_n, w_n)\}. \tag{1}$$

The utilization u_i of task τ_i is defined by (2). A proportion u_i of the total number of cycles of a core will be dedicated to executing τ_i:

$$u_i = w_i/p_i. \tag{2}$$

U, the total utilization of $\mathcal{T}$, is defined as (3):

$$U = \sum_{\forall \tau_i \in \mathcal{T}} u_i. \tag{3}$$

The processor S consists of multiple cores and is defined in (4). The nth core in S is denoted as C_n. The number of cores in S is denoted as m. Each core is assumed to have identical structure and performance. We also assume that resource sharing between the cores does not introduce any interference overhead. We have

$$S = \{C_0, \ldots, C_m\}. \tag{4}$$

F, the relative performance of S and the scaling factor for the operating clock frequency, is a number between 0 and 1. If the performance demand on S is F, then the actual frequency is the highest possible frequency of S multiplied by the factor F.

The system follows the partitioned scheduling approach; any well-known heuristic such as BFD, NFD, etc., may be adopted. The partitioned state of $\mathcal{T}$ on S is denoted $\mathcal{P}$, and the partitioned task set allocated to core C_n is denoted as $\mathcal{P}_n$. The utilization of $\mathcal{P}_n$ is defined by (5):

$$U_n = \sum_{\forall \tau_i \in \mathcal{P}_n} u_i. \tag{5}$$

For ease of description and explanation, we further define the two functions given by (6) and (7). $\Pi(\tau_i)$ gives the core that τ_i was initially partitioned into, and $\Phi(\tau_i)$ gives the

core that τ_i is currently located in. This distinction is necessary because we will dynamically migrate tasks between the cores:

$$\Pi(\tau_i) = C_j \text{ in which } \tau_i \text{ was initially partitioned,} \quad (6)$$

$$\Phi(\tau_i) = C_j \text{ in which } \tau_i \text{ is currently partitioned.} \quad (7)$$

Each partitioned task set is scheduled using EDF on its corresponding core. The performance demand of each core is decided by running the Cycle-conserving algorithm on each core individually.

To apply the Cycle-conserving algorithm, we define some dynamically updated variables. The Cycle-conserving utilization l_i of task τ_i, which is initially equal to u_i, is defined by (8). After the execution of a task, l_i is updated using the most recent actual execution time cc_i as the numerator of (2) instead of w_i. After the period p_i elapses, the task will be executed again and may now meet the worst case execution conditions; the utilization of the task will thus be reset to u_i. As a result, l_i is updated after every deadline of τ_i:

$$l_i = \begin{cases} w_i/p_i & \text{if } \tau_i \text{ is unfinished} \\ cc_i/p_i & \text{if } \tau_i \text{ is finished.} \end{cases} \quad (8)$$

L_n, the dynamic utilization of core C_n, is defined by (9). L_n is the current performance demand on C_n. Thus, as long as F is greater than L_n, all the deadlines of tasks in C_n will be met by the EDF scheduling algorithm. We will also use L to refer to the Cycle-conserving utilization of a core when the context is unambiguous. Thus, L of C_i also means L_i:

$$L_n = \sum_{\forall\, finished\ \tau_i \in \mathcal{P}_n} \frac{cc_i}{p_i} + \sum_{\forall\, unfinished\ \tau_i \in \mathcal{P}_n} \frac{w_i}{p_i}. \quad (9)$$

3.2 Power Model

The total power consumption of a CMOS-based processor consists of its dynamic power $P_{dynamic}$ and its leakage power $P_{leakage}$. We construct a processor model to evaluate the energy efficiency of the proposed algorithms.

Most of $P_{dynamic}$ is the capacitive switching power consumed during circuit charging and discharging. Generally, it is the largest part in the processor power during executing instruction. $P_{dynamic}$ can be expressed in terms of the operating voltage V_{dd}, the clock frequency f, and the switching capacity c_l as follows [23]:

$$P_{dynamic} = c_l \cdot V_{dd}^2 \cdot f. \quad (10)$$

The clock frequency f is itself related to several other factors, as given by (11). The threshold voltage V_{th} is a function of the body bias voltage V_{bs}, as seen in (12). Here, V_{th_1}, ϵ, K_1, and K_2 are constants depending on the processor fabrication technology. Generally, ϵ is between 1 and 2, so raising V_{dd} above the threshold voltage enables the processor to increase the clock frequency. In the assumed processor model, the change of f is assumed to accompany with switching V_{dd} to the lowest allowable point:

$$f = \frac{(V_{dd} - V_{th})^{\epsilon}}{L_d K_6}, \quad (11)$$

$$V_{th} = V_{th_1} - K_1 \cdot V_{dd} - K_2 \cdot V_{bs}. \quad (12)$$

TABLE 2
Constants Based on the 70 nm Technology [24]

Variable	Value	Variable	Value
K_1	0.063	I_j	4.80×10^{-10}
K_2	0.153	c_l	4.3×10^{-10}
K_3	5.38×10^{-7}	L_d	37
K_4	1.83	L_g	4×10^6
K_5	4.19	ϵ	1.5
K_6	5.26×10^{-12}	f_{min}	1×10^9
V_{bs}	-0.7	f_{max}	3×10^9
V_{th1}	0.244		

$P_{leakage}$ is caused by leakage current, which flows even while no instructions are being executed. To calculate the leakage power consumption, we adopt a processor power used in existing research [24], [25]. $P_{leakage}$ mainly consists of the subthreshold leakage current I_{subn} and the reverse bias junction current I_j. $P_{leakage}$ can be expressed as a function of these two variables, as in (13).

I_{subn} is defined by (14), where L_g is the number of components in the circuit. K_3, K_4, and K_5 are constants determined by the processor fabrication technology:

$$P_{leakage} = L_g \cdot (V_{dd} \cdot I_{subn} + \mid V_{bs} \mid \cdot I_j), \quad (13)$$

$$I_{subn} = K_3 \cdot e^{K_4 V_{dd}} \cdot e^{K_5 V_{bs}}. \quad (14)$$

The processor cores are assumed to consume both $P_{dynamic}$ and $P_{leakage}$ while executing instructions, and only $P_{leakage}$ during idle periods.

A multicore processor actually has some shared components as well, such as processor caches, buses, and so on. In this paper, we do not count the power consumption from these shared components because our goal is to reduce the power consumption of the cores themselves. The power consumption of a multicore processor is thus simply obtained by summing the power consumption of the individual cores.

The core is assumed to have two operating modes: an active state, in which it is able to execute instructions and a sleep state in which it ceases working and rests with minimized power consuming. In the sleep state, the only possible operation is a transition into the active state. In this paper, we assume that the state transition introduces no additional overhead because this factor can be treated easily in practical implementations.

In the sleep state, it is assumed that there is no $P_{dynamic}$ and that $P_{leakage}$ is 3 percent of $P_{leakage}$ at the active state with current frequency f [26].

To simulate the power consumption model just described, we adopt the realistic constants [24] given in Table 2. These constants have been scaled to 70 nm technology and are based on the original values [25] of Transmeta's Crusoe Processor which uses 180 nm technology.

By adopting the constants in Table 2, we obtain the power consumption function depicted in Fig. 3. As f decreases, the ratio of $P_{leakage}$ to P_{total} increases. Below 1.5 GHz, $P_{leakage}$ is greater than $P_{dynamic}$. $P_{dynamic}$ increases rapidly as f increases, however, and P_{total} thus rapidly increases in the high f domain.

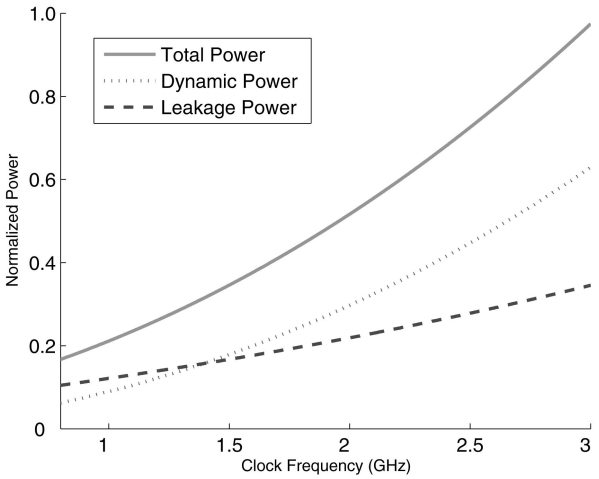

Fig. 3. Power consumption of a 70 nm core as a function of clock frequency.

4 Dynamic Repartitioning

Because $P_{dynamic} \propto f^3$, minimizing $P_{dynamic}$ in a multicore processor for a given task set is essentially the problem of generating partitioned task sets with the most balanced utilization [4]. However, even though the initial partitioned state is well-balanced the performance demand on each core changes frequently during runtime. Thus, to achieve a consistently low power consumption, the performance demand of each core must stay balanced during operation.

The intuitive way to solve the temporal unbalance is migrating some tasks on the fly from the core with high load the core with low load. To migrate tasks safely we made a simple analysis.

Notion of safe temporal task migration. At a time point T in the current period of τ_i, the remaining dynamic utilization of τ_i, u'_i, is defined as (15). cm_i is executed time of τ_i normalized to the maximum performance by the present time in this period. It shows how much additional processor performance is required to finish the remaining work of τ_i until its next deadline d_i. To be completed before d_i, the processor performance as much as u'_i should be reserved for τ_i from now to d_i:

$$u'_i = \frac{w_i - cm_i}{d_i - T}. \tag{15}$$

If a certain task τ_i is migrated from C_{src} into C_{dst} at T, then L_{src} and L_{dst} should be adjusted for matching the load change. The new L_{dst} is decided as (16). By adding u'_i, the sufficient processor cycles to finish the remaining part of τ_i before d_i will be provided to L_{dst}.

τ_i will be finished before d_i in C_{dst}, and all the additional processor cycles to process τ_i will be supplied and consumed by that time. Therefore, L'_{dst} will be same as L_{dst} after d_i:

$$L'_{dst} = \begin{cases} L_{dst} + u'_i & \text{until } d_i \\ L_{dst} & \text{after } d_i. \end{cases} \tag{16}$$

As a matter of course, L'_{dst} is larger than L_{dst} until d_i. Thus, before migrating a task, the maximum L'_{dst} should be guaranteed not to exceed 1.0. Let us define the maximum L_n from the current time to a certain time point t according to the current schedule as $M_{n,t}$. Importing τ_i into C_{dst} can be done only when $u'_i + M_{dst,d_i} < 1.0$.

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. Thus, L_{src} can be updated using (9). In this paper, we will allow the migration of a task only if it has the nearest deadline among the unfinished tasks in $\mathcal{P}_{src}$.

This migration is only effective for the current period. After the period, the migrated task should be returned to its original core. However, returning to the original core is only conceptual. If the condition can be met, the task that was executed in a foreign core can be exported into the same core again right after the next release and this will be seen as the task remains for its next period.

A migrated task may be finished earlier than its worst case execution time. It allows reducing L'_{dst} as (17), which is a combination of (16) and (9):

$$L''_{dst} = \begin{cases} L_{dst} + \frac{cc_i - cm_i}{d_i - T} & \text{until } d_i \\ L_{dst} & \text{after } d_i. \end{cases} \tag{17}$$

The migration operations can be overlapped and taken in recursive manner. In other words, a task executed in a foreign core that was migrated from the original core can be exported to the other core within the period as long as the conditions are met. Moreover, if a core has imported a task that is not completed yet, it can export multiple tasks to the other cores as long as the tasks have the nearest deadline among the unfinished tasks in the core at the exporting time.

We now suggest a dynamic approach to balancing the dynamic utilizations of each core as follows:

Notion of dynamic repartitioning. Let us assume that the partitioned state $\mathcal{P}$ has been generated as defined in Section 3.1. We further define the variables L_{max} and L_{min} as the maximum and minimum L values among all cores at calling time:

$$\forall C_n \in S, L_{max} \geq L_n, \tag{18}$$

$$\forall C_n \in S, L_{min} \leq L_n. \tag{19}$$

For each core, C_{max} and C_{min}, which are the cores that have L_{max} and L_{min}, respectively, if $\exists \tau_k$ such that $(\tau_k \in \mathcal{P}_{max}) \wedge (u'_k + L_{min} < L_{max})$, temporally migrating τ_k from $\mathcal{P}_{max}$ to $\mathcal{P}_{min}$ will lower the performance demand of processor S.

Our approach will replace $\mathcal{P}$ with $\mathcal{P}'$ that $(\mathcal{P}'_{max} = \mathcal{P}_{max} - \{\tau_k\}) \wedge (\mathcal{P}'_{min} = \mathcal{P}_{min} + \{\tau_k\}) \wedge (\forall C_n \in S - \{C_{max}, C_{min}\}, \mathcal{P}'_n = \mathcal{P}_n)$ until d_k. Under the initial partitioned state $\mathcal{P}$, all deadlines are guaranteed to be met because $\forall C_n \in S, U_n < 1$ by the assumption. If the above conditions are met, then all deadlines will also be met under $\mathcal{P}'$. As this partition is the result of migrating $\tau_k \in C_{max}$ to C_{min}, it follows that $\forall C_n \in S$, $U_n < 1$. Similarly, the repartitioned state $\mathcal{P}''$ based on $\mathcal{P}'$ is obtained by temporarily migrating a certain task τ_l from C'_{max} to C'_{min} (as determined on $\mathcal{P}'$) until d_l and $\mathcal{P}''$ also guarantees all deadlines until a certain task $\tau_l \in \{\mathcal{P}''_{max} \cup \mathcal{P}''_{min}\}$.

If we compare L_{max} (of $\mathcal{P}$) to L'_{max} (of $\mathcal{P}'$), it will always be true that $L_{max} > L'_{max}$, at least until a new period starts for one of the tasks in C'_{max}. All the cores except for C_{max}

and C_{min} remain unchanged. Thus, while using $\mathcal{P}'$ instead of $\mathcal{P}$, all the cores will be operated at the frequency conforming to L'_{max} not L_{max}. This frequency will always be lower, so the power consumption will be reduced.

Algorithm 1 describes the proposed Dynamic Repartitioning algorithm in detail. The *repartitioning* function is called whenever a task is completed or a new task period starts. Its purpose is to balance the dynamic utilizations of all cores by migrating tasks between the cores. The function migrates the task in C_{max} with the lowest required utilization at that time to C_{min}.

Algorithm 1. Dynamic repartitioning.

$\Gamma(C)$ 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} returns the core with the highest L
C_{min} returns the core with the lowest L
$M_{n,i}$ returns the maximum L_n from the calling point
  to the time i with the current schedule
$\forall m \in S$, L_m is calculated by
  (9) and (16) at every use
$\forall \tau_i \in \mathcal{T}$, d_i is updated
  to the next dead-line of τ_i at every release of τ_i
$\forall \tau_i \in \mathcal{T}$, u'_i is decided by (15)

repartitioning():
  while (true)
    $C_{src} \longleftarrow C_{max}$
    $\tau_i \longleftarrow \Gamma(C_{src})$
    $C_{dst} \longleftarrow C_{min}$
    if $((L_{dst} + u'_i < L_{src}) \wedge (u'_i + M_{dst,d_i} < 1.0))$
      $\Phi(\tau_i) \longleftarrow C_{dst}$
    else
      break

upon $task\_release(\tau_i)$:
  $\Phi(\tau_i) \longleftarrow \Pi(\tau_i)$
  repartitioning()

upon $task\_completion(\tau_i)$:
  repartitioning()

This migration process continues until the difference between the most recent calculations of L_{max} and L_{min} is less than the remaining dynamic utilization of the task currently scheduled in C_{max}. In other words, after repartitioning is complete the maximum difference between the dynamic utilizations of the cores will be less than the remaining dynamic utilization of the scheduled task in C_{max}. This approach thus restores the balance whenever the actual performance demands among cores changes significantly.

The performance demand of the source and the destination cores are corrected following (9) and (16), which are based on Cycle-conserving EDF algorithm after each migration. Before task migrations, the described safe temporal migration conditions are checked. Therefore, the schedules after calling *repartitioning* will keep all the deadlines as long as the original schedules guaranteed all the deadlines.

The suggested algorithm uses the currently scheduled task in C_{max} as a victim to be migrated. This way of selecting the victim simplifies the calculation of the performance demand in the source core of the migration into that of Cycle-conserving algorithm.

If the victim is chosen among all the unfinished task, the possibility to pick a victim successfully will be heightened. However, in those cases, the performance demand for the source core should be summing the remaining dynamic utilization values of unfinished tasks in the core. And, the performance demand update from early completed tasks should be changed based on that.

In addition to that method, if a system designer knows the characteristics of target tasks and systems in advance, there can be introduced many heuristics for picking up victims to be more efficient and more effective. Also, the task migration and scheduling overhead can be considered in a heuristic.

However, since the primary purpose of this paper is not suggesting the best heuristics algorithm, we will not describe those heuristics in detail and also the evaluation for dynamic repartitioning scheme will be done with the basic algorithm described in Algorithm 1.

5 Dynamic Core Scaling

Multicore processors have a higher energy efficiency than equivalent unicore processors because they can produce the same throughput at lower clock frequencies. This benefit, however, is primarily due to the reduction of their dynamic power consumption.

But, the situation is not same for leakage power. Leakage power increases as the scale of fabrication technology advances. Generally, a fivefold increase in leakage current, which is the source of leakage power is predicted with each technology generation [27]. And, it is proportional to the total number of circuits, which is significantly larger in a multicore processor than in a traditional processor. Thus, multicore processors and finer fabrication technologies both increase the proportion of total power consumption that can be ascribed to leakage.

The ratio of leakage power to total power also gets higher as the operating frequency decreases, because dynamic power is a cubic function of the frequency, and leakage power is a linear function. As a result, increasing the number of cores may result in a lower energy efficiency at low loads under a certain threshold, where the increased leakage power from the increased core dominates the decreased dynamic power. The relationship between power consumption, operating frequency, and number of cores is depicted in Fig. 4. While under high task loads, having more cores is always more efficient; below a certain task load, having more cores actually consumes more energy. In the assumed environment, this point lies at about 1.2 GHz or 0.4 times the maximum performance.

Most commercial multicore processors are designed with the ability to dynamically adjust the number of active cores. Each core is independently able to make a transition to any of the standard ACPI processor states [28], [29]. Some leakage power can thus be saved by simply adapting the number of active cores to the task set load at the initial stage

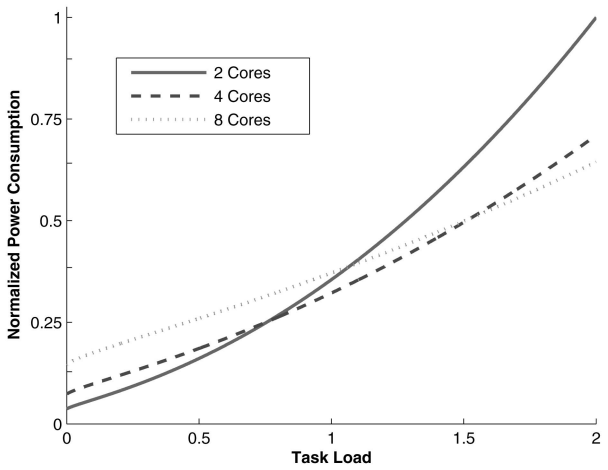

Fig. 4. Expected power consumption as a function of task load for various multicore processors.

statically. Because the dynamic utilization is always changing, however, an even higher energy efficiency can be achieved with dynamic approach. In this section, we suggest a Dynamic Core Scaling algorithm that changes the state of each core on the fly.

To determine the optimal number of cores, the relationship between task set utilization, the number of active cores, and the total power consumption must be identified. We analyze this relationship using the power model introduced in Section 3.2.

Because the execution pattern of a task is not related to the number of active cores, the relative performance F is a function only of U_{total} and the total core number m, as given by (20):

$$F = U_{total}/m. \tag{20}$$

As we are concerned with dynamic behavior, we should use the dynamic utilization instead of the static utilization:

$$F = L/m. \tag{21}$$

Furthermore, recall that the real clock frequency is expressed by (22):

$$f = F \cdot F_{max}. \tag{22}$$

Equation (23) defines the power consumption expectation function X. Its parameters are n, the number of active cores, and L, the dynamic utilization of the task set. This equation is derived from (10) and (13). V_{dd} is expressed by (24), which is derived from (11) and (12):

$$X(L,n) = n\big(c_l V_{dd}^2 f + L_g(V_{dd} I_{subn} + |V_{bs}|I_j)\big), \tag{23}$$

$$V_{dd} = \frac{(fL_d K_6)^{\frac{1}{\epsilon}} + V_{th1} - K_2 V_{bs}}{K_1 + 1}. \tag{24}$$

The function X can be used to determine the number of active cores n that minimizes X for a given L.

Equation (23) will be frequently used but is sufficiently complex that it may introduce a non-negligible overhead. For real-world implementation, we therefore recommend a hash table that maps L to the optimal number of active cores. This allows the optimal number of active cores to be decided at a constant and minimal time cost.

Returning to the main problem, note that the appropriate core number for a system is not always the power-optimal core number derived from X. It may happen that a given task set simply cannot be scheduled on the power-optimal number of cores. The problem of deciding the power-optimal core number able to schedule a given task set is of NP-hard complexity because the power-optimal scheduling problem of framed real-time task sets, which is the special instance of the problem we are dealing have NP-hard complexity [20]. Conversely, whether or not the optimal number of cores can schedule a given task set is thus also an NP-hard problem [4]. As a result, we need to find a heuristic algorithm that can obtain a near-optimal solution. We suggest the following Dynamic Core Scaling algorithm for this purpose, as defined in Algorithms 2 and 3.

Algorithm 2. Dynamic core scaling.

$\mathbb{I}$ is the set of inactivated C
$\mathbb{A}$ is the set of activated C
$\Omega(L)$ returns
  n for the lowest X under L in (23)
N_{cur} returns the number of currently activated cores
C_{exp} returns the core with the highest U in $\mathbb{I}$
C_{max} returns the core with the highest L in $\mathbb{A}$
C_{min} returns the core with the lowest L in $\mathbb{A}$
$\Gamma(C)$ returns a task τ_j such that:
  $\forall \tau_i$ where $\Phi(\tau_i) = C$,
  $(d_i \geq d_j > 0) \wedge (u'_j > 0) \wedge (\Phi(\tau_j) = C)$
$M_{n,i}$ returns the maximum L_n from the calling point
  to the time i with the current schedule
$\forall m \in S$, L_m is calculated
  by (9) and (16) at every use
$\forall \tau_i \in \mathcal{T}$, d_i is updated
  to the next deadline of τ_i at every release of τ_i
$\forall \tau_i \in \mathcal{T}$, u'_i is decided by (15)

upon $task\_release(\tau_i)$:
  if $(\Pi(\tau_i) \in \mathbb{A})$
    $\Phi(\tau_i) \longleftarrow \Pi(\tau_i)$
  else
    if $(try\_to\_migrate(\tau_i, C_{min}) = failure)$
      Activate $\Pi(\tau_i)$
      $\Phi(\tau_i) \longleftarrow \Pi(\tau_i)$
  while $(N_{cur} < \Omega(L))$
    Activate C_{exp}
  repartition()

upon $task\_completion(\tau_i)$:
  $N_{opt} \longleftarrow \Omega(L)$
  while $(N_{opt} < N_{cur})$
    if $(try\_to\_shrink() = failure)$
      break
  repartitioning()

Algorithm 3. Dynamic core scaling (continued).

```
repartitioning():
    while (true)
        C_src ⟵ C_max
        τ_i ⟵ Γ(C_src)
        C_dst ⟵ C_min
        if (L_dst + u'_i > L_src))
            break
        if (try_to_migrate(τ_i, C_dst) = failure)
            break

try_to_shrink():
    C_src ⟵ C_min
    while (∃τ_i such that Φ(τ_i) = C_src)
        τ_i ⟵ Γ(C_src)
        if (Π(τ_i) ∈ 𝔸)
            C_dst = Π(τ_i)
        else
            C_dst = C_min
        if (try_to_migrate(τ_i, C_dst) = failure)
            return failure
    Inactivate C_src

try_to_migrate(τ_i, C_m):
    if (M_{m,d_i} + u'_i < 1.0)
        Φ(τ_i) ⟵ C_m
        return success
    else
        return failure
```

The algorithm is called when a new period starts and when a task is completed. It determines the optimal core number for given L, which is newly updated with each change in the task state. If the optimal number is smaller than the current number of active cores, *try_to_shrink* will be called to reduce the number of active cores. If the optimal number exceeds the current number of active cores, the *Activate* function will be called to increase the number of active cores.

All the tasks in a core to be shrunk need to be migrated to other cores. Thus, *try_to_shrink* chooses the core with the lowest L because it will probably have the fewest tasks to migrate.

The core that was chosen as a victim since it has the lowest performance demand should migrate all the tasks scheduled in it to the other cores. Similar to *repartitioning* in Algorithm 1, choosing the task for migration is done in the earliest deadline-first order. The chosen task will be migrated into the core with the lowest performance demand at that point. The finished tasks need not to be considered in this stage since they will be migrated into the core that have the lowest performance demand at that time. The victim core will be inactivated after migrating all the unfinished tasks in it.

When *try_to_shrink* is called, it is usually the case that all the partitioned sets have low L. On many occasions, it is thus possible to migrate all the victim's tasks to other cores. There are also many conditions, however, where this is impossible. If *try_to_shrink* confirms that migration operations cannot proceed, it signals failure and stops its operation. In such cases, the algorithm gives up no more task exporting and returns to the caller. In this case, the victim core might export many tasks already. However, calling *repartitioning* after shrinking operations will remedy the problem.

An activation of a slept core occurs when the current active core number is less than the power-optimal number. The decision of whether or not to expand is made whenever L increases. It is also done when a task was newly released, and it is unable to be scheduled in C_{min}. Activation operation does not include importing tasks into the newly activated core. This will be done by *repartitioning* function, which will be called after the activation operations.

The *repartitioning* function of this algorithm, which balances the L values of the cores, is not very different from *repartitioning* function of Algorithm 1. It moves the earliest unfinished deadline task in C_{max} to the C_{src}.

As Dynamic Repartitioning, the heuristic algorithm suggested in Algorithm 2-Algorithm 3 is suggested for proving of concept. If L varies rapidly in large values, we can set a threshold L value for shrinking operation to prevent excessively frequent shrinking and expanding operations. Also known properties of a target system will help building more efficient and effective heuristic algorithms.

TABLE 3
Parameters Used in the Evaluation

Parameters	Values
α	0.3
Number of Cores (m)	4, 8 and 16
Task Load ($\frac{U}{m}$)	about 0.5 and 0.75
Ratio of cc to WCET	Uniform distribution within {0.3±0.2, 0.5±0.2 and 0.7±0.2}

6 Evaluation

The suggested algorithms were evaluated by simulating the model described in Section 3.2. The simulator randomly generates task sets with user-defined properties such as average period, average WCET, U_{total}, and so on. It can also generate initial partitions for the task sets with a variety of heuristics, including Best-Fit Decreasing (BFD), First-Fit Decreasing (FFD), Next-Fit Decreasing (NFD), and the aforementioned WFD. The results of the suggested algorithms were compared to Cycle-conserving algorithm for each task set.

There are many factors that can affect energy consumption in real-time DVS. Based on related research [13], [21], [4], [14], [7], [15], we identified the factors given in Table 3 as especially important and varied their values to simulate different situations.

The number of integrated cores is from 4 to 16. Considering that a few quad core processors are already in the market and within a decade processors with tens or hundreds cores may be shipped [30], it is close to real-world configurations.

α in Table 3 is an upper limit on the u that a task can get. The u of the tasks are randomly generated and follow a uniform distribution under α. The task load is U

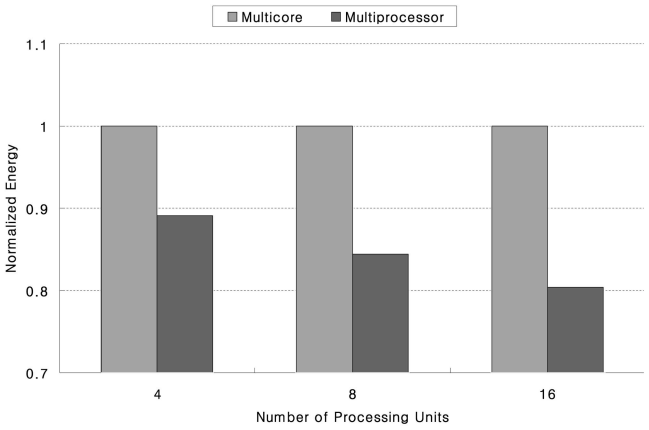

Fig. 5. Energy consumption on multicore and multiprocessor, $\text{task load} = 0.75$ and the average $cc = 0.5$.

divided by m. This refers to the relative static performance demand of a core when the tasks are partitioned into well-balanced sets.

The actual execution time of a task is generated whenever its new period starts during the simulation. It is drawn from a uniform random distribution within one of the predefined ranges given in Table 3. The benefits of the suggested algorithms are expected to depend on the relationship between actual execution times and the WCET. We thus define three different distribution ranges to obtain results under a variety of realistic operating conditions.

Before the evaluation of the suggested algorithms, as a preliminary study, we made a simple experiment to evaluate the energy loss from the shared clock frequency among cores, which is dealt in this paper. Fig. 5 shows the difference of energy consumption between a multicore system and a multiprocessor system for the same task sets. Each processor in the multiprocessor system is assumed to have equivalent performance and power consumption to a core in the multicore system. The number of processors in the multiprocessor system is same as the number of cores in the multicore system.

The difference of energy consumption grows as the number of the processing elements equipped in the system increases. About 10 percent of energy was lost with the 4-core processor and about 20 percent of additional energy was consumed with 16-core processor. It is because, as the more cores are integrated, the probability grows that only a core have exceptionally high-performance demand while those of the other cores remain low.

As mentioned earlier, in traditional multiprocessor environments, the WFD method produces the most energy-efficient partition. This result is verified by our simulations in the assumed multicore environment as well. Fig. 6 shows the results of combining different partitioning heuristics with the suggested algorithms. The energy consumption is normalized in each case to that of the WFD partition. WFD is definitely the most energy-efficient partitioning approach under the multicore system.

As expected, WFD is also the most energy-efficient partitioning approach under our Dynamic Repartitioning algorithm. The energy consumption is noticeably reduced, however, under all four initial partitions. Under low load conditions (0.5) and the Cycle-conserving algorithm, the difference between BFD and WFD is 54 percent. Under Dynamic Repartitioning, the difference between BFD and WFD is only about 15 percent. This is because the balance between the partitions was improved during runtime by Dynamic Repartitioning. Even under high load conditions, Dynamic Repartitioning conserves 28 percent of the energy consumed by the Cycle-conserving algorithm. (This number refers to the BFD partition, but the other two non-WFD partitions produce similar results.) Because WFD produces well-balanced partitions to begin with, the benefit of Dynamic Repartitioning is significantly reduced but still significant.

Considering Core Scaling under low load, there is much performance margin and for that a shrink operation successes with high probability. Thus, the number of active cores (and thus, the energy efficiency) chosen by the algorithm barely depend on the initial partition. Because during the shrink operations, many repartitioning operations are done, and the repartitioning operations are based on WFD. Compared to Cycle Conserving, Dynamic Core Scaling consumed about 40 percent less energy in cases not based on WFD and about 13 percent even in cases based on WFD.

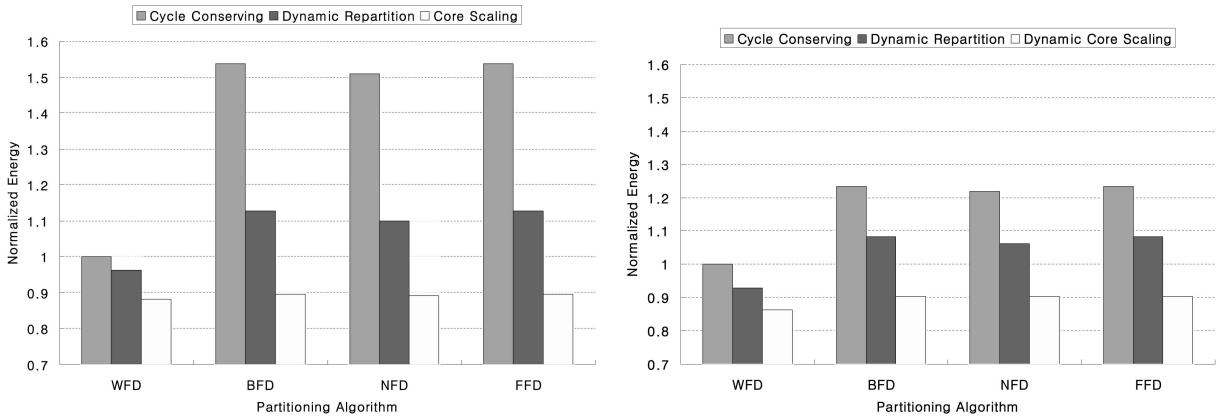

Fig. 6. Normalized energy consumption for four initial partitioning algorithms and three scheduling algorithms, at $m = 8$ and an average of $cc = 0.5$. (a) Task load: 0.5. (b) Task load: 0.75.

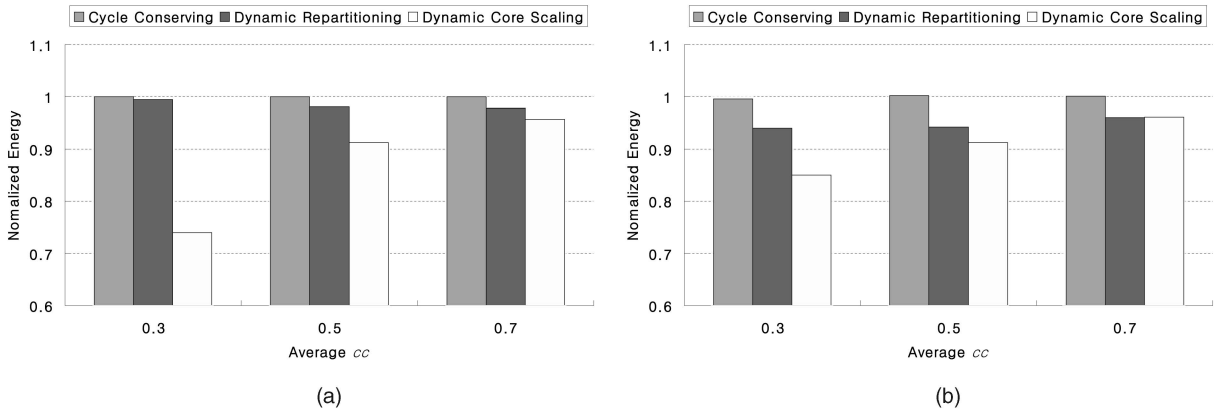

Fig. 7. Normalized energy consumption at $m = 4$. (a) Task load: 0.5. (b) Task load: 0.75.

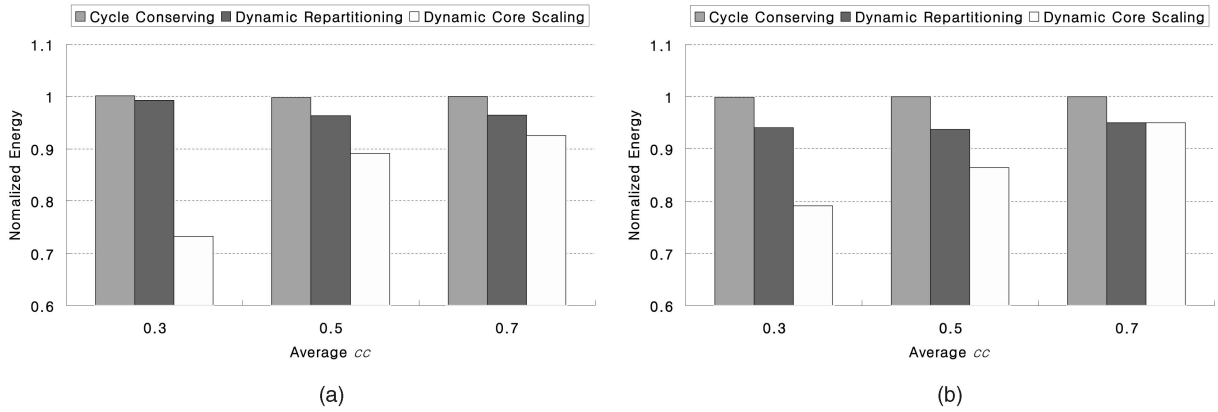

Fig. 8. Normalized energy consumption at $m = 8$. (a) Task load: 0.5. (b) Task load: 0.75.

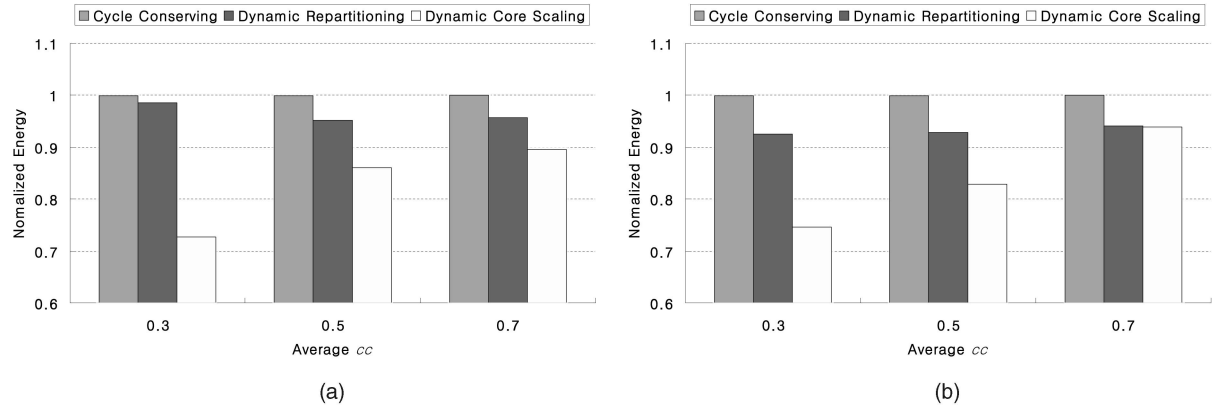

Fig. 9. Normalized energy consumption at $m = 16$. (a) Task load: 0.5. (b) Task load: 0.75.

The effect of the suggested algorithms on efficiency depends heavily on the task set load, the actual execution times, and the number of cores in the processor.

Fig. 7 presents the evaluation results of various actual execution time ranges under the two task loads in a 4-core processor. At 0.5 task load and 0.3 average cc, if all the tasks are evenly partitioned, then the L of each core will be somewhere between 0.05 and 0.5. After some time, unfinished tasks and finished tasks may coexist because the tasks have different periods. The actual L of a task thus tends to be less than 0.5. In the assumed environment, the lowest available frequency is 0.33 times the maximum frequency. Thus, when L is usually below 0.33, there may be little difference between the Cycle-conserving and Dynamic Repartitioning algorithms. This tendency is well illustrated in Figs. 7a, 8a, and 9a.

As the difference between WCET and cc becomes more important, Dynamic Repartitioning is expected to work better. This tendency is also apparent in the evaluation results when the task load is high, and the average cc is low. In the case with average $cc = 0.3$ in Fig. 7, Dynamic Repartitioning conserves 6 percent of the energy consumed by Cycle conserving.

Having more cores in a processor will lead to a greater waste of energy when the performance demands on each core are very different. The effect of Dynamic Repartitioning

therefore grows with increasing core number. The corresponding bars in Figs. 7b and 9b show that about 2 percent of the energy was saved more when $m = 16$ compared to $m = 4$. But, the difference is not as great as that obtained by varying the relation between WCET and average cc.

As expected, Dynamic Core Scaling provides its best results when the task load and average cc are low. When the task load is 0.5 and the average cc is only 0.3, 26 percent of the energy consumed by Cycle-conserving was saved (Fig. 8). When the task load and average cc are high, however, the power-optimal number of cores exceeds the number of cores in the system; Dynamic Core Scaling thus provides no additional benefit beyond Dynamic Repartitioning.

As Dynamic Core Scaling is founded on Dynamic Repartitioning, the task load has less influence on its effectiveness than the dynamic utilization. In other words, even when the task load is high, as long as the average L of the cores is low, then Dynamic Core Scaling will work well. When the task load is 0.75 and the average cc is 0.3, for example, Dynamic Core Scaling still reduced energy consumption 9 percent to 17 percent more than Dynamic Repartitioning alone.

Finally, it is apparent that increasing m strengthens the effect of Dynamic Core Scaling. More cores means a finer control over the power consumption and performance of the processor. For example, consider the case of $m = 4$, $U = 3.0$, and a task load of 0.75. Under these conditions, if any core drops into the sleep state, then U will be 1.0. In the equivalent case of $m = 16$, $U = 12$, and a task load of 0.75, dropping one core only increase the task load to 0.80—that is, by 0.05. In other words, the increase in average performance demand caused by shrinking a core goes down as the number of cores goes up. Dynamic Core Scaling thus produces better results in processors with more cores.

7 Conclusion

This paper tackles the problem of reducing power consumption in a periodic real-time system using DVS on a multicore processor. The processor is assumed to have the limitation that all cores must run at the same performance level.

To reduce the dynamic power consumption of such a system, we suggest two algorithms: Dynamic Repartitioning and Dynamic Core Scaling. The former is designed to reduce mainly the dynamic power consumption, and the latter is for the reduction of the leakage power consumption.

In the assumed environment, the best case dynamic power consumption is obtained when all the processor cores have the same performance demand. Dynamic Repartitioning tries to maintain balance in the performance demands by migrating tasks between cores during execution accompanying with deadline guarantee.

Leakage power is more important in multicore processors than in traditional unicore processors due to their vastly increased number of integrated circuits. Indeed, a major weakness of multicore processors is their high leakage power under low loads. To relieve this problem, Dynamic Core Scaling deactivates excessive cores by exporting their assigned tasks to the other activated cores.

The suggested algorithms were evaluated by simulations based on the system model and methodologies from existing research. The evaluations show that Dynamic Repartitioning can conserve up to 25 percent of the energy consumed and that Dynamic Core Scaling can conserve up to 40 percent. When the WFD algorithm is used to determine the initial task partition, Dynamic Repartitioning conserves about 8 percent, and Dynamic Core Scaling conserves about 26 percent of the energy consumed.

As mobile real-time systems grow more common, the demand for high-performance processors will also grow. It seems likely that in the future, the throughput of processors will be improved mainly by increasing the number of integrated cores. The suggested algorithms efficiently reduce both the dynamic power and leakage power of multicore processors. They are, thus, expected to be useful in mobile real-time systems of future.

On the other hand, the suggested algorithms do not take into account all the characteristics of various target task sets. In particular, if dependencies exist among the tasks or there are patterns in the actual execution times, these heuristics can be further improved by exploiting them.

Acknowledgments

This research was supported by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD) (KRF-2007-357-D00199) and also by the Ministry of Knowledge Economy of Korea under the Information Technology Research Center (ITRC) Support program supervised by the Institute of Information Technology Advancement (IITA) (IITA-2008-C1090-0801-0020).

References

[1] *Multi-Core Processors—The Next Evolution in Computing,* white paper, Advanced Micro Devices, Inc., 2005.

[2] J. Leung and J. Whitehead, "On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks," *Performance Evaluation,* vol. 2, no. 4, pp. 237-250, 1982.

[3] J.M. Lopez, M. Garcia, J.L. Diaz, and D.F. Garcia, "Worst-Case Utilization Bound for EDF Scheduling on Real-Time Multiprocessor Systems," *Proc. 12th Euromicro Conf. Real-Time Systems (ECRTS '00),* pp. 25-33, 2000.

[4] H. Aydin and Q. Yang, "Energy-Aware Partitioning for Multiprocessor Real-Time Systems," *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS '03),* p. 113b, 2003.

[5] J.M. Lopez, M. Garcia, J.L. Diaz, and D.F. Garcia, "Minimum and Maximum Utilization Bounds for Multiprocessor RM Scheduling," *Proc. 13th Euromicro Conf. Real-Time Systems (ECRTS '01),* pp. 67-75, 2001.

[6] S.K. Baruah, "Optimal Utilization Bounds for the Fixed-Priority Scheduling of Periodic Task Systems on Identical Multiprocessors," *IEEE Trans. Computers,* vol. 53, no. 6, pp. 781-784, June 2004.

[7] S. Funk, J. Goossens, and S. Baruah, "Energy Minimization Techniques for Real-Time Scheduling on Multiprocessor Platforms," Technical Report TR01-030, 1, Univ. of North Carolina, Chapel Hill, citeseer.ist.psu.edu/funk01energy.html, 2001.

[8] S. Lauzac, R. Melhem, and D. Mosse, "Comparison of Global and Partitioning Schemes for Scheduling Rate Monotonic Tasks on a Multiprocessor," *Proc. 10th Euromicro Workshop Real-Time Systems (ECRTS '98),* pp. 188-195, 1998.

[9] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *J. ACM,* vol. 20, no. 1, pp. 46-61, citeseer.ist.psu.edu/liu73scheduling.html, 1973.

[10] J. Lehoczky, L. Sha, and Y. Ding, "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior," *Proc. Real Time Systems Symp. (RTSS '89),* pp. 166-171, Dec. 1989.

[11] G. Magklis, G. Semeraro, D.H. Albonesi, S.G. Dropsho, S. Dwarkadas, and M.L. Schott, "Dynamic Frequency and Voltage Scaling for a Multiple-Clock-Domain Microprocessor," *IEEE Micro,* vol. 23, no. 6, pp. 62-68, 2003.
[12] C. Yang, J. Chen, and T. Luo, "An Approximation Algorithm for Energy-Efficient Scheduling on a Chip Multiprocessor," *Proc. Design, Automation and Test in Europe Conf. and Exhibition (DATE '05),* pp. 468-473, 2005.
[13] P. Pillai and K.G. Shin, "Real-Time Dynamic Voltage Scaling for Low-Power Embedded Operating Systems," *Proc. 18th ACM Symp. Operating Systems (SOSP '01),* pp. 89-102, 2001.
[14] H. Aydin, R. Melhem, D. Mosse, and P. Mejia-Avarez, "Power-Aware Scheduling for Periodic Real-Time Tasks," *IEEE Trans. Computers,* vol. 53, no. 5, pp. 584-600, May 2004.
[15] F. Gruian, "Hard Real-Time Scheduling for Low-Energy Using Stochastic Data and DVS Processors," *Proc. Int'l Symp. Low Power Electronics and Design (ISPLED '01),* pp. 46-51, 2001.
[16] Y. Shin, K. Choi, and T. Sakurai, "Power Optimization of Real-Time Embedded Systems on Variable Speed Processors," *Proc. IEEE/ACM Int'l Conf. Computer-Aided Design (ICCAD '00),* pp. 365-368, 2000.
[17] W. Kim, J. Kim, and S. Min, "A Dynamic Voltage Scaling Algorithm for Dynamic-Priority Hard Real-Time Systems Using Slack Time Analysis," *Proc. Conf. Design, Automation and Test in Europe (DATE '02),* p. 788, 2002.
[18] D. Duarte, N. Vijaykrishnan, M.J. Irwin, H.-S. Kim, and G. McFarland, "Impact of Scaling on the Effectiveness of Dynamic Power Reduction Schemes," *Proc. Int'l Conf. Computer Design (ICCD '02),* pp. 382-387, Sept. 2002.
[19] F. Gruian, "System-Level Design Methods for Low-Energy Architectures Containing Variable Voltage Processors," *Proc. First Int'l Workshop Power-Aware Computer Systems-Revised Papers (PACS '00),* pp. 1-12, 2000.
[20] J.-J. Chen, H.-R. Hsu, K.-H. Chuang, C.-L. Yang, A.-C. Pang, and T.-W. Kuo, "Multiprocessor Energy-Efficient Scheduling with Task Migration Considerations," *Proc. 16th Euromicro Conf. Real-Time Systems (ECRTS '04),* pp. 101-108, 2004.
[21] J.H. Anderson and S.K. Baruah, "Energy-Aware Implementation of Hard-Real-Time Systems upon Multiprocessor Platforms," *Proc. 16th Int'l Conf. Parallel and Distributed Computing Systems (PDCS '03),* pp. 430-435, 2003.
[22] M. Nikitovic and M. Brorsson, "An Adaptive Chip-Multiprocessor Architecture for Future Mobile Terminals," *Proc. Int'l Conf. Computers, Architectures, and Synthesis for Embedded Systems (CASES '02),* pp. 43-49, 2002.
[23] T.D. Burd and R.W. Brodersen, "Energy Efficient CMOS Microprocessor Design," *Proc. 28th Hawaii Int'l Conf. System Sciences (HICSS '05),* vol. 1, pp. 288-297, 1995.
[24] R. Jerjurikar, C. Pereira, and R. Gupta, "Leakage Aware Dynamic Voltage Scaling for Real-Time Embedded Systems," *Proc. 41st Ann. Technical Conf. Design Automation (DAC '04),* pp. 275-280, 2004.
[25] 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," *Proc. IEEE/ACM Int'l Conf. Computer Aided Design (ICCAD '02),* pp. 721-725, 2002.
[26] M. Fleischmann, "Longrun Power Management," technical report, Transmeta Corp., 2001.
[27] R. Jejurikar and R. Gupta, "Dynamic Slack Reclamation with Procrastination Scheduling in Real-Time Embedded Systems," *Proc. 42nd Ann. Conf. Design Automation (DAC '05),* pp. 111-116, 2005.
[28] Advanced Configuration and Power Interface, *Acpi Specification Rev. 3.0b,* http://www.acpi.info/spec.htm, 2006.
[29] 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 J.,* vol. 10, no. 2, pp. 109-122, May 2006.
[30] *Platform 2015: Intel Processor and Platform Evolution for the Next Decade,* white paper, Intel Corp., 2005.

Euiseong Seo received the PhD degree in computer science from the Korea Advanced Institute of Science and Technology. He is currently a research associate in the Computer Science and Engineering Department, Pennsylvania State University. He is affiliated with the Computer System Lab, and his research interests are in power-aware computing, real-time systems, embedded systems, and virtualization.

Jinkyu Jeong received the BS degree from the Computer Science Department, Yonsei University, and the MS degree in computer science from the Korea Advanced Institute of Science and Technology. He is currently a PhD candidate in the Computer Science Division, Korea Advanced Institute of Science and Technology. His current research interests include real-time systems, operating systems, virtualization, and embedded systems.

Seonyeong Park received the BS degree in computer science from Chungnam National University and the MS degree in computer science from the Korea Advanced Institute of Science and Technology. Currently, she is a PhD candidate in the Computer Science Division, Korea Advanced Institute of Science and Technology. She had researched at the Electronics and Telecommunications Research Institute. Her research has focused primarily on embedded file systems and ubiquitous computing services.

Joonwon Lee received the BS degree in computer science from Seoul National University in 1983 and the MS and PhD degrees from the Georgia Institute of Technology in 1990 and 1991, respectively. Since 1992, he has been a professor at the Korea Advanced Institute of Science and Technology (KAIST), Daejon, Korea. His current research interests include low power embedded systems, system software, and virtual machines.