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

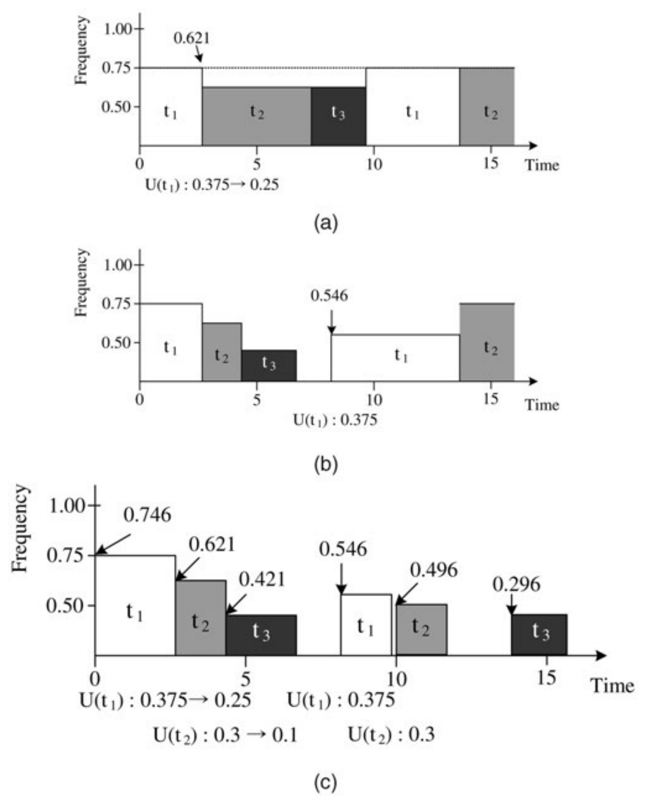

Fig. 1. Cycle-conserving algorithm on the example task set [13]. (a) After finish of executing τ_1. (b) After finish of executing τ_2 and τ_3. (c) Actual execution flow for two rounds.

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

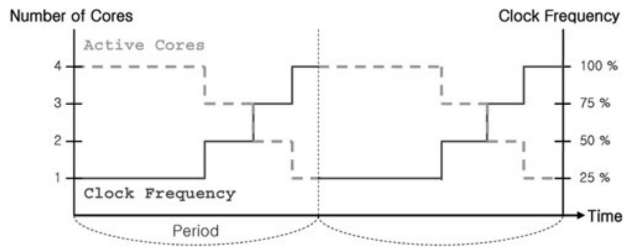

Fig. 2. Example schedule generated by heuristic algorithm [12] for DVS-CMP.

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:

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

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

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

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:

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

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

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:

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

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:

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

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]:

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

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:

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

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

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:

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

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

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.

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.

WHAT WILL YOU BUILD? | sales@docketalarm.com | 1-866-77-FASTCASE

fastcase® Smarter legal research.