Integrating Dynamic Power Management in Systems with Multiple DVS Components

Po-Liang Wu, Heechul Yun, Fatemeh Saremi, and Tarek Abdelzaher
Department of Computer Science,
University of Illinois at Urbana-Champaign,
Urbana, IL 61801
{wu87, heechul, saremi1, zaher}@illinois.edu

***Abstract*—Recent embedded computing platforms offer multiple independent clocks for different components involved in processing a single instruction stream, such as CPU and memory, giving rise to a new category of power management policies, called *MultiDVS*, where the different components can be clocked down to different degrees, independently. This paper presents the first MutliDVS scheme with dynamic power management (DPM), where the system can be put to sleep or components can be clocked down. We model power consumption in such a system, and use this model to investigate MultiDVS+DPM policies. From simulation, we show that MutliDVS+DPM can achieve up to 15% energy reduction compared to CpuDVS+DPM and 27% compared to MultiDVS alone. We also explore the impact of architectural parameters, such as scale ranges of frequencies and cache size, on efficacy of MultiDVS+DPM demonstrating both regions where it is beneficial and those where it is not.**

I. Introduction

Recent embedded platforms allow for independent adjustment of clock frequencies of the different hardware components involved in processing the same instruction stream, such as CPU and memory, thus opening new opportunities for energy savings and a new roadmap for research on embedded system energy management. This paper is the first to address effects of using dynamic power management (DPM) in such systems (henceforth called MultiDVS systems). In particular, we investigate the benefits of joint MultiDVS+DPM schemes on modern hardware, compared to DPM alone, MultiDVS alone, and the traditional (CPU only) DVS plus DPM schemes. Since many CPU-only DVS+DPM schemes are proposed in current literature, we compute an optimistic upper bound on attainable energy savings in such schemes and show that MultiDVS+DPM improves upon this bound. The paper concludes that integrating MultiDVS and DPM offers viable benefits that warrant further research.

Traditional energy management on modern hardware generally falls into Dynamic Voltage Frequency Scaling (DVS) policies[1], and Dynamic Power Management (DPM) policies. DVS reduces the active power consumption by lowering the operating frequency and voltage of the component [1], [2], [3], [4]. On the other hand, DPM can significantly reduce the power consumption by putting components into *lower-power* or *sleep* state when they are not needed [5], [6]. Our previous research [7] presented the first MultiDVS scheme that allows manipulating memory and CPU clocks separately. It demonstrated that, on currently available multiDVS systems, significant energy can be saved by clocking down memory for CPU-intensive workloads or clocking down CPU for memory-intensive ones. The current paper further demonstrates that multiDVS remains an effective scheme for energy saving when combined with dynamic power management (DPM), where the system can also be put to sleep.

In this paper, we are interested in resources that are generally needed *jointly* at a fine-grained scale (such as CPU and Memory bandwidth). We call such resources *tightly coupled*. In this case, DPM cannot be used efficiently to put one resource to sleep while the other is being used. Instead, DPM can be thought of as system-wide. This additional constraint complicates power management, since one cannot customize the amount of sleep of a resource to the utilization of that resource independently. Our investigation is thus orthogonal to those DPM schemes that consider, for example, multiple I/O peripherals that can be put to sleep independently when not in use. It is also orthogonal to work on multi-core and multiprocessor systems, where each core or processor has its own independent DVS and DPM capability. The conclusions we draw regarding energy consumption of CPU and memory remain valid for tightly-coupled resources and can be integrated in subsequent work with energy management of peripherals, as well as extended to multicore and multiprocessor systems.

Intuitively, when tightly-coupled resources, such as CPU and memory bandwidth in a system, are not equally utilized, DPM savings are a function of the bottleneck resource. For example, if either the consumed CPU or memory bandwidth is near capacity, there are limited opportunities for letting the system sleep. (Note that, we are referring to memory *bandwidth* and not its consumed size.) Beyond the limited sleep opportunity exploitable by DPM, the additional slack of the less utilized resource can be efficiently picked up by clocking it down. This additional opportunity to use DVS to exploit

[1]We shall use DVS for brevity, although technically we are considering DVFS in that both voltage and frequency can be scaled.

differences in utilization between tightly coupled resources does not arise in single DVS plus DPM systems, where the CPU is the only DVS-capable component. MultiDVS systems therefore offer new opportunities for energy savings, in which both DPM and DVS are potentially important contributors. It is these new opportunities that the current paper attempts to investigate.

The remainder of this paper is organized as follows. Section II presents related work. Section III presents the energy model. Section IV defines the energy optimization problem. Section V presents the algorithm to solve the frequency assignment problem for a set of real-time tasks and proposed a slack aggregation scheme to reduce state transition overhead. Section VI compares MutliDVS+DPM scheme to other DVS and DPM schemes. Section VII concludes the paper.

II. Related Work

A significant amount of prior literature addressed DVS [1], [2], [3], [4] and DPM [5], [6] schemes. To reduce the wake-up overhead, some work proposed aggregating slack time to achieve longer sleep intervals [8], [9].

Prior work [10], [11] suggested that the efficiency of DVS schemes is diminished because of the improving effectiveness of DPM. MultiDVS schemes [7] (which coordinate multiple tightly-coupled DVS capable components) were nevertheless shown to achieve a significant performance improvement. It is interesting to explore whether these schemes are still effective in the presence of DPM. We demonstrate, in this paper, that for tightly coupled components, such as CPU and memory, both DVS and DPM are important to help save energy. DPM alone cannot put the system to sleep even when only one component is highly utilized. In this case, MultiDVS can reduce system power consumption by reducing the frequencies of other components. Indeed, many embedded System on Chip (SoC) have tightly coupled, integrated CPU, bus, and memory components where frequency of each component is adjustable (albeit with some limitations in the frequency selection).

Combining DVS and DPM [12], [13], [14], [15], [16], [17] has been a major area for investigation. Prior work [15] proposed algorithms integrating DVS with DPM for streaming applications on embedded multiprocessors. System-wide Energy-Aware EDF (SYS-EDF) was presented [16], which integrates DPM for I/O devices and DVS for processors. A three-phase solution framework was proposed [13] to reduce system-wide power consumption with consideration to dependencies among periodic hard real-time tasks. Other work [14] developed a framework, called DFR-EDF, to integrate Device Forbidden Regions [8] with DVS. However, prior work did not consider components whose demand is coupled at the instruction or sub-instruction level (such as CPU and memory). Instead, it addressed the important complementary problem of managing loosely couple components, where each can be independently powered down.

In this paper, we propose to combine MultiDVS and DPM schemes, called *MultiDVS+DPM*, to further reduce energy consumption of systems with multiple DVS-capable (tightly-coupled) components. Our problem is different from previous work [14], [8] in the sense that we consider DPM of the entire SoC, including CPU, bus, and memory. Although the frequency of each component can be adjusted individually, the entire SoC needs to be simultaneously switched to sleep state (where CPU goes to a low power state and memory goes to a self-refresh mode). A trade-off exists between DVS and DPM schemes: DVS lowers the operating frequency but prolongs the execution time, which considerably decreases the slack time and degrades the saving efficiency of DPM. This problem is even more severe when multiple DVS and DPM capable components are involved. The technical question is how to find the frequency configuration to balance these two schemes and achieve better energy savings. This problem is NP-Hard [14], [18]. Taking real-time schedulability into consideration, we present an algorithm to find the optimal static frequency setting, and a heuristic for dynamic frequency assignment. We simulate the proposed scheme, MultiDVS+DPM, with various task sets and architecture parameters, such as scale ranges of frequencies and leakage power consumption, and compare to several other DVS and DPM schemes showing it increases savings.

III. Energy Model

In our previous work [7], we proposed an energy model, which considered multiple DVS-capable components, CPU, bus and DRAM. We employ this energy model and extend it to consider state transitions to and from sleep modes for DPM capable systems. Throughout the paper, we will use terms defined in Table I.

An important concept in considering such transitions is the *break-even time*, B. When the CPU is not busy, the system can be in either idle or sleep state. While the sleep state consumes very low power, some overhead in terms of time and energy is introduced because of the state transition between idle and sleep states. Entering sleep state saves energy only when the system remains in the sleep state for an amount of time that is equal to or larger than the break-even time, B. The break even time can be calculated as:

$$B = Max(t_r, \frac{E_r - t_r P_S}{P_I - P_S}), \quad (1)$$

where P_I is the power consumption during idle state, P_S is the power consumption during sleep state, t_r is the total time spent on switching to sleep state and back, and E_r is the total additional energy consumption spent on switching to sleep state and back [8]. Observe that in some platforms these switching times and energy overheads are asymmetric. For example, more energy may be spent on wake-up than on going to sleep. By using the totals of both transitions, the above expression remains correct regardless of symmetry or lack thereof.

For a task with a given deadline, the energy consumption can be expressed by the sum of energy in three states; pure CPU execution, cache stall, and slack. In reality, CPU execution is interleaved with cache stalls, but for the sake of computing energy, we care only for the total time spent in each state. This leads to a model that considers three consolidated blocks representing these totals, as shown in Fig. 1.

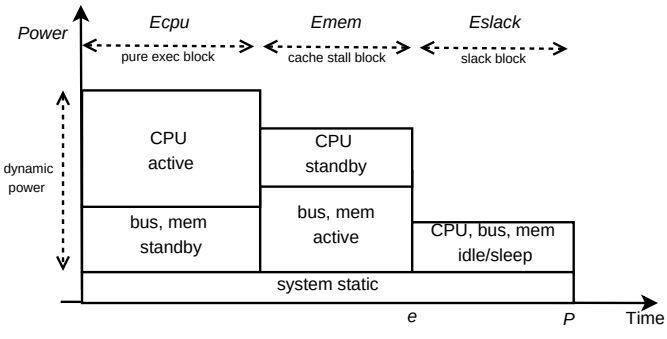

Fig. 1: Energy model for a single task with deadline (e: task finish time, P: deadline).

In the pure execution block, the CPU core executes instructions while the system bus and main memory are in standby. In the cache stall block, the cache fetches data from memory through the system bus while the CPU core is in standby, waiting for the data to become available in its cache. Therefore, the task execution time, e, can be expressed as $\frac{C}{f_c} + \frac{M}{f_m}$, where C and M are the (worst-case) total CPU and memory cycles needed, and f_c and f_m are the chosen CPU and memory frequencies, respectively. After the task finishes, the system (with its CPU and memory components) can be either left in an idle state or switched to sleep state.

Therefore, the energy consumption can be expressed by the following equation:

$$E = E_{act} + E_{slack}. \tag{2}$$

E_{act} is the energy consumption corresponding to the pure execution block and cache stall block, as shown in Equation (3). E_{slack} is energy consumption corresponding to the slack block, as shown in Equation (4). During the slack time, the system can be either in an idle or sleep state depending on whether slack is larger than the break even time, B, or not.

$$\begin{aligned} E_{act} = (K_{ca}{\cdot}V_{cpu}^N{\cdot}f_c + K_{ms}^*{\cdot}(V_{bus}^N + V_{mem}^N){\cdot}f_m + R){\cdot}\frac{C}{f_c} \\ +(K_{cs}{\cdot}V_{cpu}^N{\cdot}f_c + K_{ma}^*{\cdot}(V_{bus}^N + V_{mem}^N){\cdot}f_m + R){\cdot}\frac{M}{f_m} \end{aligned} \tag{3}$$

$$E_{slack} = \begin{cases} P_I{\cdot}(P-e), & \text{if } P-e \leqslant B \\ P_S{\cdot}(P-e-t_r) + E_r, & \text{if } P-e > B \end{cases} \tag{4}$$

TABLE I: Summary of notation.

E	total energy consumption (mJ)
e	execution time of a given task (s)
P	period(=deadline) of a given task (s)
C	CPU cycles of a given task (10^6 cycles)
M	memory cycles of a given task (10^6 cycles)
f_c	CPU clock (MHz)
f_b	system bus clock (MHz)
f_m	memory clock (MHz)
V_{cpu}	CPU voltage (V)
V_{bus}	bus voltage (V)
V_{mem}	memory voltage (V)
I	idle time dynamic power consumption of CPU, bus, and memory (mW)
R	static power consumption of the system (mW)
P_I	power consumption of idle state (mW)
P_S	power consumption of sleep state (mW)
E_r	energy spent to enter into and wakeup from sleep.
t_r	time spent to enter into and wakeup from sleep
K_{ca}	capacitance constant for active CPU (nF)
K_{cs}	capacitance constant for standby(on but idle) CPU (nF)
K_{ba}	capacitance constant for active system bus (nF)
K_{bs}	capacitance constant for standby system bus (nF)
K_{ma}	capacitance constant for active memory (nF)
K_{ms}	capacitance constant for standby memory (nF)
K_{ms}^*	$K_{ms} + K_{bs}$ when $V_{bus} = V_{mem}$ and $f_b = f_m$ (nF)
K_{ma}^*	$K_{ma} + K_{ba}$ when $V_{bus} = V_{mem}$ and $f_b = f_m$ (nF)

We validated the above model on an embedded hardware platform with an ARM926-ejs based processor. Namely, we measured actual energy consumption for different task sets and used part of the collected data to find model coefficients (via least squares regression), then used the resulting model for estimating energy consumption of the remaining data points. The experiments showed that the R^2 (the coefficient of determination) value between measured and estimated energy consumption was more than 99% and mean absolute estimation error was 1.25%, suggesting that the energy model accurately captures system behavior.

IV. System Model and Problem Definition

In this section, we formulate the problem of finding the optimal frequency setting for a set of real-time periodic tasks. Each slack interval is also determined as sleep or idle. In section V, we find an optimal static frequency assignment, in which all tasks run at the same CPU frequency, f_c, and the same memory frequency, f_m. We also propose a heuristic dynamic frequency assignment algorithm, in which the frequencies are dynamically adjusted. Moreover, in order to reduce the overhead due to state transitions, a slack time aggregation scheme is presented to combine consecutive slack intervals into a longer one without violating timing and schedulability constraints.

Consider a tasks set $T = \{T_1, T_2, ..., T_n\}$ of n periodic real-time tasks. Each task, T_i is characterized with a triple (C_i, M_i, P_i), where C_i and M_i is the number of required CPU and memory cycles in the worst case, P_i is the period, which equals to the deadline. The worst case execution time of tasks T_i is $\frac{C_i}{f_c} + \frac{M_i}{f_m}$. We consider the Earliest Deadline First (EDF) scheduling algorithm.

In order to clearly identify the relationship between frequency setting and slack time interval, we introduce a new terminology called *Execution Block*.

Definition 1: Execution Block. An execution block is the distance between the starting points of two consecutive intervals of continuous execution time.

In other words, it represents an interval of continuous execution time followed by slack time. An execution block, EB_i, is denoted by a triple, $(C_{EB_i}, M_{EB_i}, I_{EB_i})$, where C_{EB_i} and M_{EB_i} are the accumulated CPU and memory cycles and I_{EB_i} is the time interval from the start of execution to the end of slack time.

In Fig. 2, we show the execution of two tasks, T1 and T2, in a hyperperiod (defined as the least common multiple of all tasks' periods). The execution times of T1 and T2 are 2 seconds each, and the periods are 4 and 12 seconds, respectively. Therefore, there are two execution blocks, EB_1 and EB_2. EB_1 has an execution time of 6 seconds and an interval of 8 seconds. EB_2 has an execution time of 2 seconds and an interval of 4 seconds. Note that, the interval of an execution block refers to a length of time, not a repetition period.

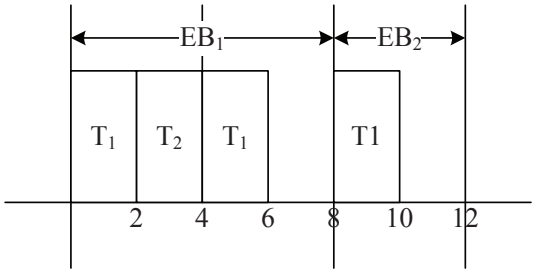

Fig. 2: Execution blocks

The slack time within an execution block can be calculated as:

$$S(EB_i) = I_{EB_i} - (\frac{C_{EB_i}}{f_c} + \frac{M_{EB_i}}{f_m}) \tag{5}$$

Depending on the amount of slack time within the execution block, it can be categorized into one of two classes: a sleep block or an idle block. If the slack time of one execution block is smaller than the break-even time, the execution block is an idle block; otherwise, it is a sleep block. Let EB_{sleep} and EB_{idle} be the set of sleep and idle execution blocks in a hyperperiod, respectively. The total energy consumption of sleep blocks consists of three parts: the active power consumption, state transition overhead, and the sleep energy consumption during slack time, as shown in Equation (6).

$$E_{sleep} = \sum_{EB_i \in EB_{sleep}} (E_{act,i} + \cdot E_r + (S(EB_i) - t_r) \cdot P_S). \tag{6}$$

Similarly, the total energy consumption of idle blocks is:

$$E_{idle} = \sum_{EB_i \in EB_{idle}} (E_{act,i} + S(EB_i) \cdot P_I) \tag{7}$$

Therefore, the system-wide energy consumption is:

$$E_{sys} = E_{sleep} + E_{idle} \tag{8}$$

The most basic optimization formulation of combined MultiDVS and DPM under EDF scheduling is:

$$\text{Minimize } E_{sys} \tag{9}$$

$$\text{Subject to } \sum_{T_i \in T} \frac{\frac{C_i}{f_c} + \frac{M_i}{f_m}}{P_i} \leqslant 1 \tag{10}$$

Equation (9) minimizes energy consumption in a hyperperiod. Equation (10) is the necessary and sufficient schedulability constraint [19] for *work-conserving* scheduling. The above basic formulation has two important limitations. First, it assumes that a single frequency value is selected for each component (although different components, such as CPU and memory, may operate at different frequencies) throughout the hyperperiod. We call such a frequency assignment, *static*. Second, it assumes that scheduling is work-conserving. Hence, it does not consider consolidating slack times into bigger gaps (that are bigger than the break even point) to increase opportunities for sleep. These limiting assumptions will be relaxed later in the paper.

V. Frequency Assignment and Slack Aggregation

In this section, we first present a simple solution to the above basic formulation for static frequency assignment and work-conserving scheduling (in Section V-A). We then extend the scheme to dynamic frequency assignment and non-work-conserving scheduling. For dynamic frequency assignment (Section V-B), we present a heuristic for selecting a different frequency for each execution block. For non-work-conserving scheduling (Section V-C), we present a slack aggregation algorithm which delays some execution blocks in a way that allows consolidating slack times to minimize wakeup overheads. We make sure to do so without violating schedulability constraints such that schedulability conditions are not affected.

A. Static Frequency Assignment

Consider the simplest case where component frequencies are fixed and scheduling is work conserving. Let us consider only CPU and memory components, although extensions to other components are straightforward.

The key in computing the energy corresponding to a frequency assignment is to understand, for the given assignment, when the processor can go to sleep. Once those

intervals are determined, Equation (8) can be used to compute energy consumption. To determine sleep intervals, we compute the amount of slack that remains in each execution block from Equation (5). For a given block and a given component frequency assignment, there are three possibilities:

- The slack is greater than the break-even time, B. This means one can put the system to sleep and save energy. As mentioned earlier, we call such a block a sleep block.
- The slack is equal or less than the break-even time. This means that there is not enough time to sleep because the cost of the sleep/wakeup transition is not subsumed by savings. This block remains idle during slack time. We call it an idle block.
- The slack is zero or negative. This means that the frequency was lowered enough that the block should be merged with its chronologically following one. The above steps are then repeated for the merged block.

The outcome of this algorithm (for a given frequency assignment), is a categorization of each block into idle or sleep. Since most current systems have only two or three independently clocked components involved in the execution of an instruction stream (e.g., CPU and memory components, or CPU, memory and I/O bus components), each having only a few discrete frequencies, it is easy to enumerate all feasible frequency combinations exhaustively. Thus, starting with the set of blocks that results at the maximum frequency assignment for all components, we apply the above three steps, for each frequency combination, one block at a time, to identify which blocks are sleep blocks and which are idle blocks. The energy can then be computed from Equation (8) and the optimal frequency assignment is trivially obtained. Algorithm V.1 summaries the steps.

This assignment assumes work-conserving scheduling (i.e., task execution is never delayed, for example, to consolidate slack intervals). It also assumes that frequency assignment is static. In the following, we relax these limitations.

B. Dynamic Frequency Assignment

In this section, we present a heuristic for dynamic frequency assignment, which independently assigns component frequencies for the duration of each execution block. We evaluate the resulting performance improvement and compare with other schemes in Section VI.

As before, the heuristic algorithm starts by identifying the set of execution blocks based on the maximum CPU and memory frequency. The next step is to execute the technique presented above to one block only. We can thus find the locally optimal frequency setting for that one block. To simplify, we do not explore frequencies that lead to block overflow. In other words, if a frequency assignment leads to overflow, it is ignored. Once the optimal frequency for a block is found, we repeat for the next block, until a frequency is assigned to each. Algorithm V.2 summaries the steps.

Algorithm V.1 Static Frequency Assignment Algorithm

1: Identify execution blocks based on maximum CPU frequency f_c^{max} and maximum memory frequency f_m^{max}.
2: $E^{opt} \leftarrow ENERGY\_MAX$
3: **for** each feasible combination of frequencies, f_c, f_m **do**
4: **for** each execution block EB_i **do**
5: **if** $S(EB_i) \geqslant B$ **then**
6: Mark EB_i as a sleep block
7: **else if** $0 < S(EB_i) < B$ **then**
8: Mark EB_i as an idle block
9: **else**
10: Merge EB_i to EB_{i+1}
11: **end if**
12: **end for**
13: Calculate the energy consumption,E, based on the execution block configuration
14: **if** $E < E^{opt}$ **then**
15: $E^{opt} \leftarrow E, f_c^{opt} \leftarrow f_c, f_m^{opt} \leftarrow f_m$
16: **end if**
17: **end for**
18: Return f_c^{opt} and f_m^{opt}

Algorithm V.2 Dynamic Frequency Assignment Algorithm

1: Identify execution blocks based on maximum CPU frequency f_c^{max} and maximum memory frequency f_m^{max}.
2: **for** each execution block EB_i **do**
3: $E^{opt} \leftarrow ENERGY\_MAX$
4: **for** each feasible combination of frequencies, f_c, f_m **do**
5: **if** $S(EB_i) \geqslant B$ **then**
6: Mark EB_i as a sleep block
7: **else if** $0 < S(EB_i) < B$ **then**
8: Mark EB_i as an idle block
9: **else**
10: Continue
11: **end if**
12: **end for**
13: Calculate the energy consumption,E
14: **if** $E < E^{opt}$ **then**
15: $E^{opt} \leftarrow E, f_{c,i}^{opt} \leftarrow f_c, f_{m,i}^{opt} \leftarrow f_m$
16: **end if**
17: Return the list of frequency settings
18: **end for**

C. Slack Aggregation Policy

In this section, we describe a heuristic for slack aggregation to increase slack length beyond the break-even point and hence increase opportunities to sleep. The question is how to determine the amount of allowable delay for each execution block without violating its deadline constraints. Since EDF scheduling can achieve 100% utilization, the execution of each job corresponding to one task, T_i, can be delayed by $(1-U)\cdot P_i$, where U is the sum of utilization of each task and P_i is the period of task T_i. This is proven by the following lemma.

Lemma 1: Given total utilization U, the jobs corresponding to a task T_i can be delayed by $(1-U)\cdot P_i$ under EDF scheduling.

Proof: We consider the delay as part of execution and add it to the execution time of T_i. The total utilization then becomes $U+((1-U)\cdot P_i)/P_i = 1$. Since the EDF scheduling can achieve 100% utilization, the taskset is still schedulable. ■

Based on this observation, we introduce the term *Feasible Delay*, which is the amount of delay can be applied to an execution block without causing any job to miss its deadline. Suppose there are n tasks, and EB_i contains m tasks ($m \leqslant n$). Note that, one task can have multiple jobs in EB_i. Let U be the total utilization of n tasks. The feasible delay of an execution block, EB_i, can be calculated as follows:

$$FeasibileDelay_i = \min(S(EB_i), \min_{j\in m}((1-U)*P_j)) \quad (11)$$

Recall that $S(EB_i)$ is the slack time of EB_i.

Lemma 2: The execution of one execution block can be delayed by the amount of feasible delay without making any job violate its deadline constraint.

Proof: If the feasible delay is applied to an execution block, the execution of all jobs are postponed by the same amount. Since the feasible delay is the minimum of slack time and the tolerable delay of all jobs, no job will miss its deadline if the execution is postponed by the feasible delay. ■

According to the feasible delay concept, we present a heuristic for slack aggregation. For two consecutive execution blocks, the slack aggregation is triggered if one of the following rules is satisfied.

1) The first block is an idle block, and applying the feasible delay to the second one makes the first block a sleep block.
2) The feasible delay of the second block equals to its slack time, and the sum of the slack time of two execution blocks is lager than the break even time. In other words, the slack time of two execution blocks is integrated as a single sleep interval.

Algorithm V.3 summaries the steps.

Note that, the concept of execution block can augment other slack time aggregation schemes, such as, device forbidden region [8]. However, in Section VI, we show that further potential improvement is less than 10% in most cases. This is attained by comparing to an ideal case where sleep and wakeup have no cost. Hence, the length of the break even time is zero. Any interval of slack can be used for sleep. No slack consolidation scheme can increase the amount of sleep time or decrease the cost of switching beyond the above ideal case. Therefore, comparing to the ideal case, we bound all achievable improvements over our scheme that can be attained by more clever slack consolidation and non-work conserving scheduling.

Algorithm V.3 Slack Aggregation Algorithm

```
1: U ← Σ_{i=1}^{n} (C_i/T_c + M_i/T_m)/P_i
2: for each execution block EB_i do
3:    FeasibileDelay_i ← S(EB_i)
4:    for each task, T_j, in EB_i do
5:       if (1 − U) · P_j < FeasibileDelay_i then
6:          FeasibileDelay_i ← (1 − U) · P_j
7:       end if
8:    end for
9: end for
10: for two consecutive execution blocks EB_i and EB_{i+1} do
11:    if EB_i is idle AND S(EB_i)+FeasibileDelay_{i+1} > B then
12:       Delay the execution of EB_{i+1}
13:    end if
14:    if FeasibileDelay_{i+1} = S(EB_{i+1}) AND S(EB_i)+ S(EB_{i+1}) > B then
15:       Delay the execution of EB_{i+1}
16:    end if
17: end for
```

VI. Evaluation

In this section we evaluate the energy savings of the proposed static and dynamic MultiDVS+DPM schemes with and without slack aggregation using simulation. In particular, we are interested in understanding how much benefit we can gain using MultiDVS+DPM schemes compared to single component DVS schemes (with or without DPM), DPM schemes (without MultiDVS) and MultiDVS schemes (without DPM).

A. Model Validation

We first describe how we obtained model parameters used in the simulation. All parameters were obtained from a real ARM926 based hardware platform shown in Fig. 3.

The total energy is the sum of active energy E_{act} and slack energy E_{slack} (see Equation (3) and Equation (4)). The part E_{act} does not depend on DPM. As such, we obtained and validated the parameters of this part of the model in our earlier work on MultiDVS [20]. We briefly summarize this experiment here for completeness. We first measured energy consumption of four synthetic programs

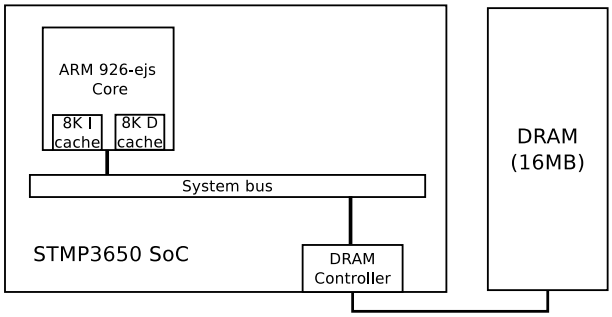

Fig. 3: Tested hardware platform. DRAM uses a fixed voltage (3.0V) while CPU and system bus share a common varying voltage. The DRAM operates at system bus frequency.

TABLE II: Model parameters for active energy consumption

Capacitance (nF)			
K_{ca}	K_{cs}	K^*_{ma}	K^*_{ms}
0.52	0.30	0.18	0.05

(each with a different cache stall ratio: 0%, 10%, 25%, and 55%). For each program, we used eight different clock configurations. We measured energy consumption for the 32 different configurations. Then we performed nonlinear least square regression to determine the value of each parameters in E_{act}. The results is shown in Table II. Finally, we performed validation by computing the prediction error in energy consumption, which was shown to be less than 1.25%.

Computing E_{slack} required new experiments, since this energy depends on the newly added DPM scheme. To compute E_{slack}, we measured three parameters P_I, P_S (total system power consumption during idle and sleep period), and t_r (sleep transition time overhead) directly from the board. The sleep transition is performed as a single system call in our platform. We configured the system call so that it goes to sleep and immediately wakes-up. We measured the total time for ten pairs of transitions, and averaged it to get the t_r. Notice that we could not measure E_r directly due to the short duration of the event which can not be accurately captured by our measurement equipment. We instead calculated it from our energy equation by multiplying the maximum power consumption of the system by the measured time, t_r.

B. Simulation Setup

We performed a set of simulation experiments to evaluate the effectiveness of our proposed MultiDVS+DPM schemes using the parameters obtained from the previous section.

TABLE III: Model parameters for sleep and idle energy consumption

P_I (mW)	P_S (mW)	E_r (mJ)	t_r(ms)
77.70	6.52	1110	2

We use average power consumption, calculated as the total energy consumption in a hyperperiod divided by the length of the hyperperiod, as the evaluation metric. We further normalize the average power consumption to that of the MAX scheme (where all components operate at the top frequency) in every figure unless mentioned otherwise. For each point, we randomly generated 100 different tasksets at the given overall utilization and cache stall ratio. [2]. Task period was randomly chosen between 20ms to 1300ms as in [8]. Each point in the following figures is the average of the 100 tasksets. We used EDF schedule for all simulations.

C. Evaluated Algorithms

Thirteen schemes are compared in our evaluation (some are omitted in some figures). We briefly introduce them below:

Static individual schemes: (1) MAX: Tasks are executed with the maximum CPU, 200MHz, and maximum memory frequency, 100MHz. During the slack time, it consumes idle power(P_I) and does not go to sleep. (2) CpuDVS: It scales CPU frequency, but memory frequency is set to maximum. (3) MultiDVS: It is the scheme described in [7]. We assign optimal CPU and memory frequency but do not go to sleep during slack time. (4) DPM: It is the same as MAX except that the CPU and memory go to sleep state when there is no task to run and the slack is larger than break-even time; they wake up on a task invocation.

Static combination schemes: (5) CpuDVS+DPM: It jointly manages CPU frequency and sleep time. (6) MultiDVS+DPM: It is the proposed scheme described in Section V. (7) MultiDVS+DPM+Aggr: It is the same as the previous scheme except it also aggregates adjacent execution blocks whenever possible to consolidate sleep.

Dynamic combination schemes: (8) CpuDVS+DPM_Dynamic: The scheme is dynamic in that it can change the CPU frequency for each execution block separately. (9) MultiDVS+DPM_Dynamic: It is the same as the previous scheme except it adjusts both CPU and memory frequency.

Ideal static and dynamic schemes: (Ideal means that there is no sleep overhead in both time t_r and energy E_r). (10) CpuDVS+DPM-Ideal: It is the same as CpuDVS+DPM except the sleep overhead is zero; While unrealistic, we can think of this scheme as an upper bound of energy saving of any aggregation scheme. No slack aggregation scheme (e.g., DFR and CEED [8], [16]) can outperform this one. (11) MultiDVS+DPM-Ideal: It assumes that sleep overhead is zero, hence this is an upper bound of any slack aggregation for the MuliDVS+DPM combination. (12) CpuDVS+DPM_Dynamic-Ideal: It is the same as CpuDVS+DPM_Dynamic except the sleep overhead is zero. This is an upper bound on the best slack aggregation for CpuDVS+DPM in a dynamic setting. (13)

[2]We define the cache stall ratio, r, as $r = \frac{M}{C+M}$, where C is the CPU cycles and M is memory cycles [7]

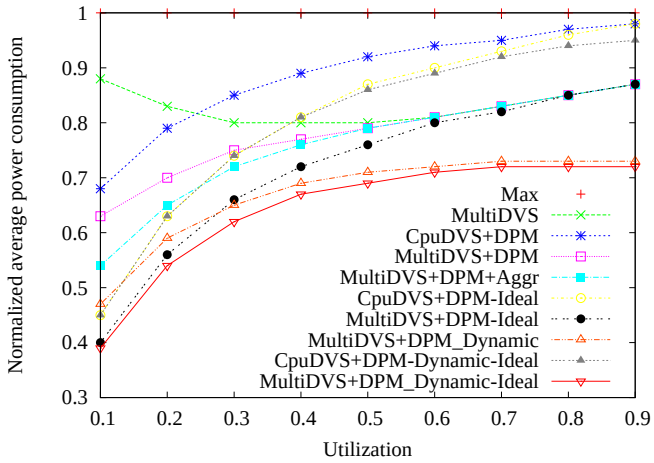

Fig. 4: Average power consumption with varying utilization and constant cache stall ratio = 0.01.

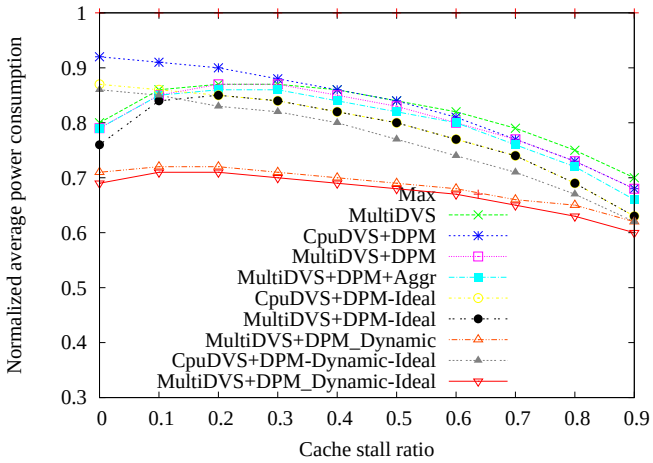

Fig. 5: Average power consumption with varying cache stall ratio and constant utilization = 0.5.

MultiDVS+DPM_Dynamic-Ideal: It is the same as MultiDVS+DPM_Dynamic but the sleep overhead is zero.

D. Varying Taskset Utilization

Fig. 4 shows the average power consumption of the compared schemes for varying utilization while the cache stall ratio is fixed at 0.01, representing a highly CPU-intensive load. Such load might be constructed by design in that almost all instructions and data fit in a local (dedicated) cache. As the utilization increases, the effectiveness of all schemes generally decreases. DPM saves more energy when utilization is low because there is enough slack time to compensate for the overhead of sleep. As the utilization increases, however, the effectiveness of DPM decreases compared to other schemes due to the decreased slack time. CpuDVS scheme saves very little energy, because the taskset is CPU intensive and reducing CPU frequency does not save energy due to the significantly increased execution time. In comparison, MultiDVS consistently saves energy over all utilization values. The saving mostly comes from reducing bus and memory power consumption because the taskset is CPU intensive in this experiment.

The MultiDVS+DPM scheme takes advantage of both DPM and MultiDVS and performs better than other schemes (except the MultiDVS+DPM_Ideal and the aggregation scheme for obvious reasons) in both static and dynamic schemes. The dynamic scheme has better performance than the static. In comparison, another combined scheme, CpuDVS+DPM, performs worse and is almost identical to the DPM scheme. Again, this is because for CPU intensive tasks, lowering CPU frequency does not save energy much. Comparing MultiDVS+DPM+Aggr scheme to MultiDVS+DPM_Ideal, the additional saving of more clever aggregation scheme is less than 10% (except at utilization 0.1) and quickly decreases as utilization increases. In addition, the gap between MultiDVS+DPM_Dynamic and MultiDVS+DPM_Dynamic-Ideal is generally less than 5%, which shows limited performance improvement of slack aggregation.

It is important to compare the MultiDVS+DPM scheme with CpuDVS+DPM-Ideal for both static and dynamic cases, because it is the upper bound of power saving for all CpuDVS+DPM schemes. Starting from utilization 0.3, the MultiDVS+DPM scheme does always better than CpuDVS+DPM-Ideal. This means that the new scheme beats the best that CpuDVS+DPM can ever offer. This motivates investigating better MultiDVS+DPM schemes. Similarly, the dynamic scheme, MultiDVS+DPM_Dynamic, outperforms CpuDVS+DPM_Dynamic-Ideal when the utilization is larger than 0.2. Therefore, both static and dynamic cases confirm the advantage of MultiDVS in conjunction with DPM.

E. Varying Cache Stall Ratio

Figure 5 shows the average power consumption of the compared schemes for varying cache stall ratio while the overall utilization is fixed at 0.5. The DPM scheme is not affected by stall ratio changes because utilization, hence the execution time of tasks, is the same. The MultiDVS scheme saves energy at both low and high cache stall ratio. This is because when the stall ratio is low, and hence load is CPU intensive, it saves energy by lowering memory frequency. When stall ratio is high, and hence load is memory intensive, it saves energy by lowering CPU frequency. CpuDVS also save energy but only when the cache stall ratio is high, hence the load is memory intensive. The CpuDVS+DPM scheme is similar to the CpuDVS scheme. Also, the energy saving trends of MultiDVS+DPM are similar to MultiDVS. This is expected from the previous section, because the usefulness of DPM diminishes at higher utilization. With

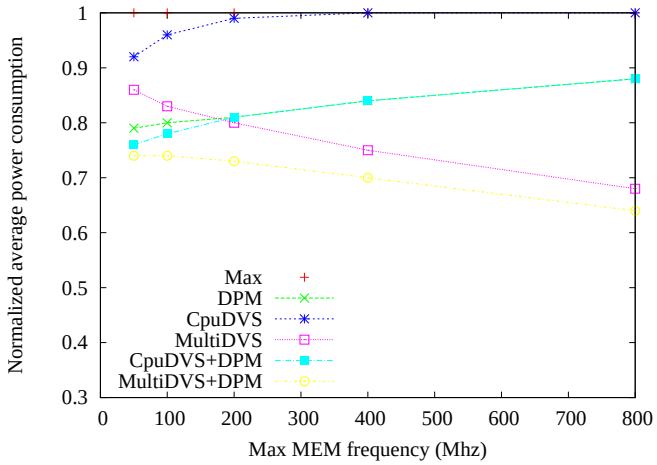

Fig. 6: Varying memory frequency range. stall ratio=0.1 and utilization=0.2

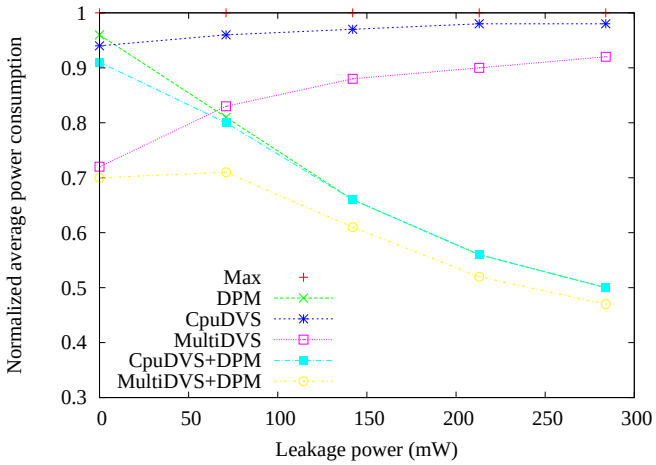

Fig. 7: Varying leakage power. stall ratio = 0.01, utilization = 0.2

a utilization of 0.5, in this experiment, performance of MultiDVS+DPM is not significantly improved over MultiDVS. This is true for all cache stall ratios. Both MultiDVS+DPM+Aggr and Ideal save more energy compared to MultiDVS+DPM, but the difference is within a five percent range. Again, CpuDVS+DPM-Ideal is an upper bound on the best that slack aggregation can do. MultiDVS+DPM is better than that bound, especially when the cache stall ratio is low (0.0 - 0.1). A similar observation holds true for dynamic schemes; MultiDVS+DPM_Dynamic always outperforms CpuDVS+DPM_Dynamic-Ideal. This means no dynamic slack aggregation algorithm for CpuDVS and DPM can do better than MultiDVS+DPM_Dynamic.

F. Future Architecture Change

In all the previous evaluation, we used system parameters that are obtained by regression from a real hardware platform. In this section, we explore how the results may differ if different hardware platforms were used. In doing so, we are interested in applying current trends in mobile processor technologies to evaluate the effectiveness of the proposed and existing schemes in future hardware platforms.

We investigated three architectural parameters that can affect energy consumption: (1) CPU/Memory frequency range, (2) Cache size, (3) Leakage power.

First, the frequency of CPU and memory are increasing in general and are expected to remain that way in the near future. Fig. 6 shows the impact of increasing maximum memory(bus) frequency. As memory frequency increases, MultiDVS becomes relatively more effective compared to the DPM scheme. The CpuDVS+DPM can not adapt to memory frequency changes. MultiDVS+DPM scheme, in comparison, is better than any individual policy because MultiDVS+DPM exploits the increased memory frequency range as well as the benefit of DPM. The effect of CPU frequency increase is similar to Fig. 6. CpuDVS+DPM is similar to MultiDVS+DPM because CpuDVS can get benefit from the increased CPU frequency range.

Second, the cache size is generally increasing. This effectively means that the stall ratio will decrease for the same workload. This can be indirectly seen in Fig. 5. As the stall ratio decreases, MultiDVS+DPM scheme do considerably better than CpuDVS+DPM scheme. Note that, the savings of CpuDVS+DPM scheme become close to DPM only. This indicates that the effectiveness of CPU-only DVS schemes will diminish in the future. The same observation is also presented by a recent empirical study [10] on server processors.

Third, the leakage power is generally increasing. As the leakage power increases, DPM becomes more effective while MultiDVS becomes less effective, as shown in Fig. 7. The results show an opposite direction for the effectiveness of DPM and MultiDVS compared to Fig. 6.

Based on these different architectural effects, we can predict that the proposed scheme, MultiDVS+DPM, will outperform any individual schemes, DPM, MultiDVS, as well as CpuDVS+DPM, with relative advantages of these schemes yet to be determined.

G. Evaluation Summary

Since the effectiveness of the compared DVS and DPM schemes depends on both taskset characteristics and system architecture parameters, we thoroughly considered a variety of different settings and possible architecture effects. Simulation results have shown that the proposed combination scheme, MultiDVS+DPM, outperforms individual schemes, CpuDVS, MultiDVS, and DPM, in all situations. It also outperforms the CpuDVS+DPM scheme by more than 10% in CPU centric workloads. The reason is that when tasks are CPU centric, scaling CPU frequency will significantly increase the execution time and thus limit effectiveness, while memory frequency can be greatly reduced with limited execution time impact. Moreover, in a tightly coupled

system, since the system cannot be switched often to sleep state when some component has high utilization, the effectiveness the DPM is limited. Therefore, the proposed MultiDVS+DPM scheme achieves better energy savings for both static and dynamic schemes across a wide range of parameters. Finally, comparing MultiDVS+DPM+Aggr with the corresponding Ideal scheme, we can clearly see the the additional performance gain is limited even if more clever slack aggregation algorithms are applied. Hence, the current heuristic is already sufficiently close to optimal savings.

VII. Conclusions and Future Work

In this study, we proposed a combination of MultiDVS and DPM scheme to reduce system-wide energy consumption for real-time systems. In the energy model, we focused on three DVS and DPM capable components, CPU, bus, and memory. A solution was then derived to find frequency assignments for multiple components considering system constraints. Based on the model and the solution, we first presented a combination scheme to find optimal static frequencies for multiple components for periodic real-time tasks. In addition, we further exploited the benefit of dynamic frequency assignment and slack aggregation schemes. The proposed schemes are compared with other DVS and DPM schemes to demonstrate its effectiveness. In short, MutliDVS+DPM can achieve up to 15% energy reduction compared to CpuDVS+DPM and 27% compared to MultiDVS alone.

One important avenue for extension is to further exploit the usage of the proposed scheme to other platform. For one thing, multi-core or multi-processor systems offer multiple components with DVS capability. What are the energy and computation models to reflect this kind of system? How much benefit we can get when taking task migration/aggregation into consideration? For another, sever farms are one of the major sources for energy consumption, and energy efficiency for server farms becomes an important issue. It would be interesting to explore the potential usage to deploy the proposed scheme in the server farm environment.

Acknoledgements

The work reported in this paper was funded in part by NSF grant CNS 09-16028. Finding and opinions expressed are those of the authors and do not represent those of the funding agencies.

References

[1] P. Mejia-Alvarez, E. Levner, and D. Mossé, "Adaptive scheduling server for power-aware real-time tasks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 3, no. 2, p. 306, 2004.

[2] X. Zhong and C.-Z. Xu, "Energy-Aware Modeling and Scheduling of Real-Time Tasks for Dynamic Voltage Scaling," *Proceedings of IEEE RTSS*, pp. 10 pp.–375, 2005.

[3] H. Aydin, R. Melhem, D. Mossé, and P. Alvarez, "Determining optimal processor speeds for periodic real-time tasks with different power characteristics," in *Proceedings of ECRTS*, 2001, pp. 225–232.

[4] R. Jejurikar and R. Gupta, "Optimized slowdown in real-time task systems," *IEEE Transactions on Computers*, vol. 55, no. 12, p. 1588, 2006.

[5] L. Benini, A. Bogliolo, and G. De Micheli, "A survey of design techniques for system-level dynamic power management," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 8, no. 3, pp. 299–316, 2002.

[6] B. Brock and K. Rajamani, "Dynamic power management for embedded systems," in *Proceedings of the IEEE SOC Conference*, 2003.

[7] H. Yun, P. Wu, A. Arya, T. Abdelzaher, C. Kim, and L. Sha, "System-Wide Energy Optimization for Multiple DVS Components and Real-Time Tasks," in *Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on*. IEEE, 2010, pp. 133–142.

[8] V. Devadas and H. Aydin, "Real-time dynamic power management through device forbidden regions," in *Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium-Volume 00*. IEEE Computer Society Washington, DC, USA, 2008, pp. 34–44.

[9] D. Zhu, R. Melhem, and B. Childers, "Scheduling with dynamic voltage/speed adjustment using slack reclamation in multiprocessor real-time systems," *IEEE Transactions on Parallel and Distributed Systems*, pp. 686–700, 2003.

[10] E. Le Sueur and G. Heiser, "Dynamic voltage and frequency scaling: The laws of diminishing returns," *Workshop on Power Aware Computing and Systems*, 2010.

[11] A. Miyoshi, C. Lefurgy, E. Van Hensbergen, R. Rajamony, and R. Rajkumar, "Critical power slope: Understanding the runtime effects of frequency scaling," in *Proceedings of the 16th international conference on Supercomputing*. ACM, 2002, p. 44.

[12] V. Devadas and H. Aydin, "On the interplay of dynamic voltage scaling and dynamic power management in real-time embedded applications," in *Proceedings of the 8th ACM international conference on Embedded software*. ACM, 2008, pp. 99–108.

[13] P. Rong and M. Pedram, "Power-aware scheduling and dynamic voltage setting for tasks running on a hard real-time system," in *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*. IEEE Press, 2006, pp. 473–478.

[14] V. Devadas and H. Aydin, "DFR-EDF: A Unified Energy Management Framework for Real-Time Systems," in *2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2010, pp. 121–130.

[15] H. Liu, Z. Shao, M. Wang, and P. Chen, "Overhead-aware system-level joint energy and performance optimization for streaming applications on multiprocessor systems-on-chip," in *Proceedings of ECRTS*, 2008, pp. 92–101.

[16] H. Cheng and S. Goddard, "Integrated device scheduling and processor voltage scaling for system-wide energy conservation."

[17] T. Simunic, L. Benini, A. Acquaviva, P. Glynn, and G. De Micheli, "Dynamic voltage scaling and power management for portable systems," in *Proceedings of the 38th annual Design Automation Conference*. ACM, 2001, p. 529.

[18] X. Zhong and C. Z. Xu, "System-Wide Energy Minimization for Real-Time Tasks: Lower Bound and Approximation," *The International Conference on Computer-Aided Design*, pp. 516–521, 2006.

[19] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[20] H. Yun, P. Wu, A. Arya, T. Abdelzaher, C. Kim, and L. Sha, "System-Wide Energy Optimization for Multiple DVS Components and Real-Time Tasks (Extended journal version)," *Real-time System Journal (to appear)*, 2011.