A Near-optimal Solution for the Heterogeneous Multi-processor Single-level Voltage Setup Problem

Tai-Yi Huang, Yu-Che Tsai and Edward T.-H. Chu

National Tsing Hua University
Dept. of Computer Science
Hsinchu, Taiwan 300, R.O.C
{tyhuang, yctsai, edward}@eos.cs.nthu.edu.tw

Abstract

A heterogeneous multi-processor (HeMP) system consists of several heterogeneous processors, each of which is specially designed to deliver the best energy-saving performance for a particular category of applications. A low-power real-time scheduling algorithm is required to schedule tasks on such a system to minimize its energy consumption and complete all tasks by their deadline. The problem of determining the optimal speed for each processor to minimize the total energy consumption is called the voltage setup problem. This paper provides a near-optimal solution for the HeMP single-level voltage setup problem. To our best knowledge, we are the first work that addresses this problem. Initially, each task is assigned to a processor in a local-optimal manner. We next propose a couple of solutions to reduce energy by migrating tasks between processors. Finally, we determine each processor's speed by its final workload and the deadline. We conducted a series of simulations to evaluate our algorithms. The results show that the local-optimal partition leads to a considerably better energy-saving schedule than a commonly-used homogeneous multi-processor scheduling algorithm. Furthermore, at all measurable configurations, our energy consumption is at most 3% more than the optimal value obtained by an exhaustive iteration of all possible task-to-processor assignments. In summary, our work is shown to provide a near-optimal solution at its polynomial-time complexity.

1. Introduction

A heterogeneous multi-processor (HeMP) system places on a single system a set of heterogenous processors. Each processor may have its own instruction set architecture (ISA) specially designed to provide the best energy-saving performance for a particular category of applications. The HeMP architecture is commonly adopted by a low-power embedded system on which several categories of applications are hosted. Examples are embedded devices hosting multimedia applications [14] or applications that demand audio signal processing [12]. Many of these low-power HeMP systems are also real-time systems [15] in each task must complete its execution by its deadline to avoid any critical failure. For this reason, the problem of minimizing the energy consumption without missing any deadline has become an important issue in constructing low-power real-time HeMP systems.

Several low-power real-time scheduling algorithms [16, 7] have been proposed to address this problem. The work of [16] schedules a set of independent periodic tasks on a HeMP system in which each processor has a finite number of speeds, each of which is driven by a corresponding voltage. A similar problem is solved by [7] for a system where each processor has a fixed speed (voltage). Both algorithms assume that their processor speeds are known as a priori. Accordingly, their schedules may not be optimal in minimizing energy consumption. On the other hand, several other algorithms [3, 8, 13] have been designed to determine the number of levels and the optimal speed for each level to achieve the minimum energy consumption. It is called the *voltage setup* problem [8]. The single-level voltage setup problem is solved by [3] to deal with a system where a processor has one speed. The multi-level voltage setup problem is addressed by [8] and [13] to deal with the case where a processor has multiple speeds. However, all these work [3, 8, 13] focus on a one-processor system and cannot be applied to solve the multi-processor voltage setup problem.

In this paper, we provide a near-optimal solution for the HeMP voltage setup problem. To our best knowledge, our work is the first one that addresses this problem. The discussed workload consists of n frame-based real-time tasks to be scheduled on m heterogeneous processors. All tasks are independent and non-preemptible. We focus our discussion on the single-level problem where each processor has only one speed. This problem can be formulated as a non-linear generalized assignment problem (GAP) that is proven to be NP-hard. In other words, an optimal solution requires exponential time complexity.

We provide a couple of polynomial-time solutions to determine each processor's speed such that the total energy consumption is minimized. Initially, each task is assigned to a processor in a local-optimal manner. We next propose a greedy-based method to migrate tasks out of an overloaded processor in order to reduce energy. This method selects only one task during each migration. We further improve its performance by selecting a group of tasks on the same processor in each migration. A dynamic-programming method is proposed to avoid redundant computations. Finally, we determine each processor's speed by its final workload and the deadline. The greedy-based method has $O(nm \log n)$ time complexity. The dynamic-programming method has $O(nmX)$ time complexity, where X bounds the load of the most-loaded processor after the initial partition.

We conducted a series of simulations to evaluate our algorithms. Our simulations model a set of off-the-shelf embedded processors including ARM processors and TI DSP processors. For comparison, we also implemented a commonly-used homogeneous multi-processor (HoMP) low-power algorithm called list scheduling [11, 5]. Without any energy-reduction method, the list scheduling algorithm delivers the worst performance. The combination of list scheduling and our dynamic-programming energy-reduction method, however, still consumes considerably more energy than the combination of the local-optimal task partition and the dynamic-programming method. This result shows the importance of initial task assignments and the effectiveness of our local-optimal partition. Each experimental result is compared to the optimal value obtained by an exhaustive iteration of all possible task-to-processor assignments. At all measurable configurations, our energy consumption is at most 3% more than the optimal value. These results well demonstrate that our work provides a near-optimal solution for the HeMP single-level voltage setup problem.

The rest of this paper is structured as follows. Section 2 describes the system model and the local-optimal task partition. Section 3 presents the greedy-based energy-reduction algorithm. The DP-based energy-reduction algorithm is described in Section 4. Section 5 presents our performance analysis. Finally, Section 6 concludes this paper and discusses future work.

1.1 Related Work

A number of real-time scheduling algorithms have been proposed for a HoMP system [2, 1, 4]. The Proportionate-fair (Pfair) algorithm, proposed by Baruah *et al.* [2], provides an optimal real-time schedule for periodic tasks. This algorithm, however, considers no energy consumption and cannot be used in a low-power system. Anderson *et al.* [1] proposed a method for finding the optimal number of processors on which a given set of periodic tasks incurs the minimum energy consumption. Chen *et al.* [4] optimally bounds the energy consumption for a set of frame-based tasks, each of which has different power characteristics. All these algorithms focused their discussion on HoMP systems. Without considering that a task may have different execution times on heterogeneous processors, these algorithms cannot be directly applied on HeMP systems.

Yu *et al.* [16] proposed a low-power real-time algorithm to schedule a set of independent periodic tasks on HeMP systems in which each processor is capable of dynamic voltage scaling (DVS). In other words, tasks running on the same processor may be executed at different speeds. This problem is formulated as a linear GAP and a linear relaxation heuristic solution is provided. Assuming that the available processor speeds are known as a priori, this algorithm provides a schedule that minimizes energy under this constraint. Hsu *et al.* [7] addressed this problem for a HeMP system in which each processor has a fixed speed. This problem is formulated as an integer linear programming problem and a polynomial-time approximation solution is provided. Again, this algorithm assumes that each processor speed is given as a constraint. As a result, their real-time schedule may not be optimal in reducing energy without such a constraint.

The voltage setup problem is first formulated in [8] to determine the number of levels and at which values should voltages be implemented to deliver the optimal energy-saving performance for a specific application. Aydin *et al.* [3] proved that the optimal voltage for a one-processor single-level problem is equal to its utilization when the maximum speed is normalized to one. Hua *et al.* [8] proposed an analytical solution for a one-processor two-level problem and Seo *et al.* [13] proposed an optimal solution for a one-processor multi-level problem. To the best of our knowledge, our work is the first one that addresses the multi-processor voltage setup problem. Because of the popularity in HeMP embedded systems nowadays, our study started with the HeMP single-level problem.

2 System Model

Our work adopts a commonly-used multi-processor model consisting of m processors sharing a common memory [6]. We assume a HeMP system in which each processor may have its own ISA. The discussed workload is a set of frame-based real-time tasks [10, 6]. Our goal is to determine a speed for each processor such that the total energy consumption required to complete all tasks before their deadline is minimized.

2.1 Energy Model and Task Model

Each processor assumes a commonly-used energy model where a processor speed is almost linearly related to its supply voltage and the power consumption of a processor increases cubically with its processor speed [9, 6]. Let $C_1, C_2, \ldots, C_m$ denote these m processors. We use P_j to denote the power consumption of C_j at the speed of S_j. Thus,

$$P_j = k_j \times S_j^3, \tag{1}$$

where k_j is an adjusted switched capacitance of C_j.

We adopt a frame-based real-time task model in which a frame of length D is executed repeatedly. We use $\mathcal{T}$ to denote a set of n real-time tasks, $\tau_1, \tau_2, \ldots, \tau_n$, to execute within each frame. Each task τ_i is released at the beginning of a frame and must complete its execution by end of this frame. All tasks are independent and non-preemptible. Because of its periodicity, we only consider the problem of scheduling $\mathcal{T}$ in a single frame.

Each task τ_i may be complied against more than one ISA and can be executed on a set of heterogenous processors. The workload of a task in our model is denoted by its cycle count, instead of its execution time. Let $x_{i,j}$ denote the number of clock cycles to execute τ_i on processor C_j. If τ_i cannot be executed on C_j, $x_{i,j}$ is set to infinite. Let $\mathcal{T}_j$ denote the set of tasks scheduled to be executed on C_j, and X_j denote the sum of cycle counts of these tasks. That is,

$$X_j = \{ \sum x_{i,j} \mid \tau_i \in \mathcal{T}_j \}. \tag{2}$$

Once $\mathcal{T}_j$ is determined, because all tasks must complete their execution by D, we calculate the processor speed S_j of C_j by $S_j = \frac{X_j}{D}$. The power consumption P_j of C_j is therefore obtained by

$$P_j = k_j \times (\frac{X_j}{D})^3.$$

Finally, let E_j denote the energy consumption of C_j in one frame. We have

$$\begin{aligned} E_j = P_j \times D &= k_j \times (\tfrac{X_j}{D})^3 \times D \\ &= \tfrac{k_j \times X_j^3}{D^2}. \end{aligned} \tag{3}$$

2.2 Problem Formulation

Following Eq (3), we define $F_{i,j}$ as an index of energy consumption to execute τ_i on each processor C_j, $j = 1$ to m,

$$F_{i,j} = k_j \times x_{i,j}^3.$$

The smaller $F_{i,j}$ is, the less energy consumption τ_i incurs on C_j. In addition, for each task τ_i we define $\alpha_i = (\alpha_{i,1}, \alpha_{i,2}, \ldots, \alpha_{i,m})$ as a list of all processor numbers, sorted by $F_{i,j}$ in ascending order. In other words,

$$F_{i,\alpha_{i,j}} \leq F_{i,\alpha_{i,j+1}}, \text{ for } j = 1 \text{ to } m-1.$$

We use car(α_i) to denote the first entry of α_i and cadr(α_i) to denote the second entry of α_i. We call car(α_i) as the most-favored processor of τ_i on which τ_i incurs the least energy consumption and cadr(α_i) as its secondly-favored processor.

We use F_j to denote an index of the total energy consumption on the processor C_j,

$$F_j = k_j \times X_j^3,$$

where X_j is the sum of cycle counts of all tasks scheduled on C_j, defined in Eq. (2). We use $\mathcal{E}$ to define the total energy consumption of all processors. By Eq. (3), we have

$$\mathcal{E} = (\sum_{i=1}^{m} F_i)/D^2.$$

Our problem can be therefore formulated as a non-linear GAP problem that minimizes $\mathcal{E}$. Finally, we define γ as a list of all processor numbers, sorted by F_j in descending order. That is, $\gamma = (\gamma_1, \gamma_2, \ldots, \gamma_m)$,

$$F_{\gamma_j} \geq F_{\gamma_{j+1}}, \text{ for } j = 1 \text{ to } m-1.$$

2.3 *kX*³-based Task Partition

We initially partition all n tasks onto m processors in a local-optimal manner. Algorithm 1 summarizes this process of task partition. This algorithm, called **kX³-Partition**, first constructs α_i for each task τ_i (line 3 to 5). It next assigns a task τ_i to its most-favored processor (line 6 to 9). Finally, we calculate each X_i's and F_i's and construct the γ list (line 10 to 13). As no load balancing is considered, some processors may be favored by many tasks and become overloaded. A couple of solutions will be presented later to balance loads among processors in order to reduce the total energy consumption.

Table 1 shows an example of 5 tasks on a 3-processor system. The k_i index of C_1, C_2, and C_3 are 1×10^{-6}, 2×10^{-6}, and 3×10^{-6} (mW/Hz^3), respectively. The cycle

	k_1		k_2		k_3		D
mW/Hz^3	1×10^{-6}		2×10^{-6}		3×10^{-6}		0.05 (s)
	$x_{i,1}$	$F_{i,1}$	$x_{i,2}$	$F_{i,2}$	$x_{i,3}$	$F_{i,3}$	$min(F_{i,j})$
τ_1	**10**	1×10^{-3}	30	5.4×10^{-2}	10	3×10^{-3}	1×10^{-3}
τ_2	30	2.7×10^{-2}	**10**	2×10^{-3}	40	1.92×10^{-1}	2×10^{-3}
τ_3	80	5.12×10^{-1}	50	2.5×10^{-1}	**10**	3×10^{-3}	3×10^{-3}
τ_4	80	5.12×10^{-1}	**20**	1.6×10^{-2}	20	2.4×10^{-2}	1.6×10^{-2}
τ_5	**30**	2.7×10^{-2}	60	4.32×10^{-1}	70	1.029	2.7×10^{-2}
total	40	6.4×10^{-2}	30	5.4×10^{-2}	10	3×10^{-3}	48.4 (mJ)

Table 1: A 5-task 3-processor example

Algorithm 1

```
1: Procedure kX^3-Partition()
2: initialize all data structures to $\emptyset$;
3: for all $\tau_i$ do
4:     construct $\alpha_i$;
5: end for
6: for $i = 1$ to $n$ do
7:     $j$ = car($\alpha_i$);
8:     add $\tau_i$ to $\mathcal{T}_j$;
9: end for
10: for $i = 1$ to $m$ do
11:     calculate $X_i$ and $F_i$;
12:     $\gamma$ = InsertReverseSorted($\gamma$, $i$, $F_i$);
13: end for
```

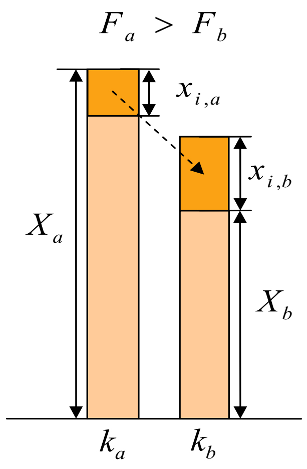


Figure 1: The migration of τ_i from C_a to C_b

counts of τ_1 on C_1, C_2, and C_3 are 10, 30, and 10. Accordingly, $F_{1,1}, F_{1,2}, F_{1,3}$ are 1×10^{-3}, 5.4×10^{-2}, and 3×10^{-6}. Thus, $\alpha_1 = (1, 3, 2)$. Similarly, $\alpha_5 = (1, 2, 3)$. Algorithm 1 assigns τ_1 to C_1 as it incurs the least $F_{1,j}$. In addition, $\tau_2, \tau_3, \tau_4, \tau_5$ are assigned to C_2, C_3, C_2, C_1, respectively. This partition results in a total energy consumption of 48.4 mJ when $D = 0.05$ (second). Finally, $\gamma = (1, 2, 3)$ as $F_1 = 6.4 \times 10^{-2}$, $F_2 = 5.4 \times 10^{-2}$, and $F_3 = 3 \times 10^{-3}$.

3 A Greedy-Based Energy-Reduction Algorithm

We present here a greedy-based method to select tasks for migration. In the following, we first describe an index to sort tasks in a processor by their potential contribution of reducing $\mathcal{E}$ when being migrated. We next present the core algorithm and its timing complexity.

3.1 Migration Order

We migrate τ_i from its currently-assigned processor to another processor, if such a migration results in a smaller $\mathcal{E}$. Let C_a denote the processor τ_i is currently assigned to. Let C_b denote the target processor. Without loss of generosity, we assume that $F_a > F_b$, as shown in Figure 1. We use $x_{i,a}$ and $x_{i,b}$ to denote the workload τ_i incurs on both processors. By Eq. (3), a smaller $\mathcal{E}$ is achieved by migrating τ_i from C_a to C_b if and only if

$$k_a \times \frac{(X_a - x_{i,a})^3}{D^2} + k_b \times \frac{(X_b + x_{i,b})^3}{D^2} \leq k_a \times \frac{X_a^3}{D^2} + k_b \times \frac{X_b^3}{D^2}.$$

This equation can be simply induced to

$$\frac{k_a}{k_b} \geq \frac{(X_b + x_{i,b})^3 - X_b^3}{X_a^3 - (X_a - x_{i,a})^3}. \tag{4}$$

Let $|\mathcal{T}_i|$ denote the number of tasks in $\mathcal{T}_i$. For each C_i, we define $\beta_i = (\beta_{i,1}, \beta_{i,2}, \ldots, \beta_{i,|\mathcal{T}_i|})$ as a list of task numbers where each task $\tau_{\beta i,j} \in \mathcal{T}_i$. Initially, because C_i is the most-favored processor for $\tau_{\beta_{i,j}}$, i = car($\alpha_{\beta_{i,j}}$) for any j. We sort β_i by its impact on the total energy reduction when a task migration takes place. In other words, by migrating $\tau_{\beta i,j}$ to its next favored processor, we expect to reduce more on $\mathcal{E}$ than migrating $\tau_{\beta i,j+1}$.

Let C_a denote the most-loaded processor. Following Eq. (4), we define a dynamic index Δ_i for each task τ_i to denote the reduction impact on $\mathcal{E}$ if τ_i is migrated from C_a to its next favored processor C_b,

$$\Delta_i = k_a \times (X_a^3 - (X_a - x_{i,a})^3) - k_b \times ((X_b + x_{i,b})^3 - X_b^3).$$

The larger Δ_i is, the more energy reduction it brings by migrating τ_i to its next favored processor. Ideally, we can sort β_a by each task's Δ_i in descending order. However, X_a and X_b change after each migration. Accordingly, maintaining a sorted list of β_a by Δ_i incurs significant run-time computational overhead to update each task's Δ_i and re-sort β_a after each migration. To reduce this overhead, we define a static index δ_i,

$$\delta_i = \frac{k_a \times x_{i,a}}{k_b \times x_{i,b}} \tag{5}$$

to replace Δ_i, as Δ_i grows positively with k_a and $x_{i,a}$ and negatively with k_b and $x_{i,b}$. Because δ_i is defined by constants only, its run-time computational overhead is significantly reduced. Instead of maintaining β_a by Δ_i, we sort β_a by δ_i in descending order. That is,

$$\delta_{\beta a,j} \geq \delta_{\beta a,j+1}, \quad \text{for } j = 1 \text{ to } |\mathcal{T}_a| - 1. \tag{6}$$

β_a now defines the migration order of tasks on C_a.

3.2 The Greedy-Based Algorithm

Algorithm 2 summarizes the **Greedy-Based** method. We first call **kX^3-Partition** to partition tasks and construct α and γ lists (line 2). We next calculate δ_i by Eq. (5) for each τ_i and construct β_i by Eq. (6) for each C_i (line 3 to 8). The task-migration process always takes place on the most-loaded processor (*i.e.,* the first entry of γ) (line 9 and line 24). Let C_i be the most-loaded processor. For tasks scheduled on C_i, we follow the order given in β_i to migrate tasks. Let τ_j denote the task with the largest δ_j (line 11) and C_k denote its next favored processor (line 12). If τ_j satisfies Eq. (4), we simply call **MigrateTask**, defined in Algorithm 3, to migrate τ_j from C_i to C_k (line 14). Otherwise, we skip C_k and try to migrate τ_j to it next favored processor (line 16 to 19). If no target processor is available, we simply skip τ_j and pick the next task in β_i for migration (line 21). Finally, when no task can be migrated out of the most-loaded processor, this algorithm stops (line 10).

We use the schedule given in Table 1 to illustrate how **Greedy-Balanced** works. Initially, $\gamma = (1, 2, 3)$ indicates that C_1 is the most-loaded processor. For tasks scheduled on C_1, because $\delta_1 = 1/3$ and $\delta_5 = 1/4$, we migrate τ_1 from C_1 to its next favored processor C_3. The updated γ list shows

Algorithm 2

```
1: Procedure Greedy-Based()
2: kX$^3$-Partition(); // initial task partition
3: for all $\tau_i$ do
4:   $\delta_i$ = CalculateDelta(car($\alpha_i$), cadr($\alpha_i$)); // by Eq. (5)
5: end for
6: for all $C_i$ do
7:   construct $\beta_i$ by Eq. (6);
8: end for
9: $i$ = car($\gamma$); // the most-loaded processor
10: while $\beta_i \neq \emptyset$ do
11:   $j$ = car($\beta_i$); // migrating $\tau_j$ out
12:   $k$ = cadr($\alpha_j$); // a possible target processor
13:   if ($\tau_j$ satisfies Eq. (4)) then
14:     MigrateTask($C_i$, $\tau_j$, $C_k$);
15:   else
16:     $\alpha_j$ = cons(car($\alpha_j$), cddr($\alpha_j$)); // removing $C_k$ from $\alpha_j$
17:     if (cdr($\alpha_j$) $\neq \emptyset$) then
18:       $\delta_j$ = CalculateDelta(car($\alpha_j$), cadr($\alpha_j$)); // by Eq. (5)
19:       $\beta_i$ = InsertSorted(cdr($\beta_i$), $\delta_j$);
20:     else
21:       $\beta_i$ = cdr($\beta_i$); // skipping $\tau_j$ and considering the next task
22:     end if
23:   end if
24:   $i$ = car($\gamma$); // the most-loaded processor
25: end while
```

that C_2 becomes the most-loaded processor. However, because both τ_2 and τ_4 cannot satisfy Eq. (4), our migration process stops here. Table 2 shows the final schedule of this example. The total energy consumption becomes 42 mJ, reduced from the original 48.4 mJ. Finally, we determine the processor speeds of C_1, C_2, and C_3 at 600, 600, and 400 Hz, respectively.

We use a binary heap to construct each α, β, and γ list, as most references to these lists are limited within the first and the second elements. Inserting or deleting an element into a binary heap of n elements takes $O(\log n)$ time. In addition, referencing an element by car() or cadr() is done in constant time. For **kX^3-Partition**, we take $O(nm \log m)$ to construct all α lists and take $O(m \log m)$ to construct γ. Thus, **kX^3-Partition** is bounded by $O(nm \log m + m \log m)$. For **Greedy-Based**, we take $O(n \log n)$ to construct all β lists. Because there are at most n tasks to be migrated, each migration may target at m different processors, and each migration incurs a constant number of heap insertions and deletions, we bound **Greedy-Based** by $O(nm \log n)$.

4 A DP-Based Energy-Reduction Algorithm

The **Greedy-Based** algorithm follows its β list to migrate tasks. When a task's migration fails to reduce $\mathcal{E}$, the migration process of this processor stops. In other words, **Greedy-Based** migrates one task at a time and its migration order is strictly limited by its β list. We remove this

	k_1		k_2		k_3		D
mW/Hz^3	1×10^{-6}		2×10^{-6}		3×10^{-6}		0.05(s)
	$x_{i,1}$	$F_{i,1}$	$x_{i,2}$	$F_{i,2}$	$x_{i,3}$	$F_{i,3}$	
τ_1	10	1×10^{-3}	30	5.4×10^{-2}	**10**	3×10^{-3}	P_3
τ_2	30	2.7×10^{-2}	**10**	2×10^{-3}	40	1.92×10^{-1}	P_2
τ_3	80	5.12×10^{-1}	50	2.5×10^{-1}	**10**	3×10^{-3}	P_3
τ_4	80	5.12×10^{-1}	**20**	1.6×10^{-2}	20	2.4×10^{-2}	P_2
τ_5	**30**	2.7×10^{-2}	60	4.32×10^{-1}	70	1.029	P_1
total	30	2.7×10^{-2}	30	5.4×10^{-2}	20	2.4×10^{-2}	
	600Hz	216mW	600Hz	432mW	400Hz	192mW	42 (mJ)

Table 2: The greedy-based energy-reduction for the example shown in Table 1

restriction here and select a group of tasks on the same processor for migration. We avoid redundant computations by a dynamic-programming method.

4.1 The Recursive Formula

Let C_a denote the most-loaded processor. Let $Z = |\mathcal{T}_a|$ denote the number of tasks initially assigned to C_a by **kX^3-Partition**. X_a is used to denote the sum of cycle counts of all tasks in $\mathcal{T}_a$. In addition, we construct β_a as described in Section 3.1 to sort tasks in $\mathcal{T}_a$ by their δ_i's defined in Eq. (5). Let $\mathcal{A}$ denote the maximum amount of reduction in $\mathcal{E}$ by migrating a group of tasks out of C_a. To determine this group of tasks, we further define $M[k, g]$ as the maximum amount of reduction by migrating a group of tasks, each of which is one of the first k tasks in β_a and the sum of cycle counts of all migrated tasks is less than or equal to g. Obviously, we have

$$\mathcal{A} = \max_{0 \le g \le X_a} \{M[Z, g]\}. \tag{7}$$

In addition, we have an initial setting of $M[0, g] = 0$ for $g = 0$ to X_a.

Because each migration changes the workload of both the source and the target processors, we define $H[k, g]$ to record the workload of each processor after the group of tasks selected in $M[k, g]$ are migrated. We represent $H[k, g]$ as a list of m entries and $H[k, g][i]$ denotes its i-th entry, the workload of C_i. Let X_i denote the initial workload of C_i after **kX^3-Partition**. We have,

$$H[0, g] = \{X_1, X_2, \ldots, X_m\}, \text{ for } g = 0 \text{ to } X_a.$$

The k-th task in β_a may or may not be selected to be migrated in $M[k, g]$. For simplicity, we use η to denote $\beta_{a,k}$. When τ_η is not selected, we have $M[k, g] = M[k-1, g]$. Otherwise, when τ_η is migrated, we have

$$M[k, g] = M[k-1, g-x_{\eta,a}] + \textbf{EnergyDelta}(H[k-1, g-x_{\eta,a}], \eta),$$

Algorithm 3

```
1: Procedure MigrateTask($C_i$, $\tau_j$, $C_k$)
2: $\alpha_j$ = cdr($\alpha_j$); $\beta_i$ = cdr($\beta_i$);
3: $\delta_j$ = CalculateDelta(car($\alpha_j$), cadr($\alpha_j$));
4: $\beta_k$ = InsertSorted($\beta_k$, $\delta_j$);
5: $X_i = X_i - x_{j,i}$; update $F_i$;
6: $X_k = X_k + x_{j,k}$; update $F_k$;
7: $\gamma$ = ReverseSorted($\gamma$);
```

where **EnergyDelta**$(H[k, g], \eta)$ denotes the amount of reduction in $\mathcal{E}$ by migrating τ_η out of C_a at the workload of $H[k, g]$. In summary, we define $M[k, g]$ by

$$M[k, g] = \max \left\{ \begin{array}{l} M[k-1, g], \\ M[k-1, g-x_{\eta,a}]+ \\ \textbf{EnergyDelta}(H[k-1, g-x_{\eta,a}], \eta) \end{array} \right\}. \tag{8}$$

The **EnergyDelta** algorithm is shown in Algorithm 4. We first determine the energy reduction by migrating τ_η out of C_a (line 3). We next calculate the increase of energy in its target processor (line 10), obtained by the next element in α_η. If we do not have any reduction in $\mathcal{E}$ (line 11), we continue to the next processor in α_η (line 6). This algorithm stops either when we locate a target processor that results in a positive reduction in $\mathcal{E}$ or when we reach the end of α_η.

Finally, we need to document the change of processor workloads in $H[k, g]$. If τ_η is not selected to be migrated in $M[k, g]$, we have $H[k, g] = H[k-1, g]$. Otherwise, we first make a copy of $H[k-1, g-x_{\eta,a}]$ to $H[k, g]$. We next change two entries in $H[k, g]$ by

$$\begin{array}{l} H[k, g][a] = H[k, g][a] - x_{\eta,a}; \\ H[k, g][b] = H[k, g][b] + x_{\eta,b}; \end{array} \tag{9}$$

where b is the target processor determined in **EnergyDelta**.

Algorithm 4

```
1: Procedure **EnergyDelta**$(L,\ \eta)$
2: $a = \mathrm{car}(\alpha_\eta)$; $p = \alpha_\eta$;
3: Minus $= k_a L[a]^3 - k_a (L[a] - x_{\eta,a})^3$;
4: $R = 0$;
5: while $R \le 0$ do
6:    if $((p = \mathrm{cdr}(p)) == \mathrm{NULL})$ then
7:       break;
8:    end if
9:    $b = \mathrm{car}(p)$;
10:   Plus $= k_b (L[b] + x_{\eta,b})^3 - k_b L[b]^3$;
11:   $R =$ Minus $-$ Plus;
12: end while
13: return $R$;
```

Algorithm 5

```
1: Procedure **MaxReduction**$(a)$
2: for $g = 0 \ldots X_a$ do
3:    $M[0,g] = 0$; $H[0,g] = \{X_1, X_2, \ldots, X_m\}$;
4: end for
5: for $k = 1 \ldots Z$ do
6:    for $g = 0 \ldots X_a$ do
7:       $\eta = \beta_{a,k}$;
8:       if $(g < x_{\eta,a} \parallel (M[k-1, g-x_{\eta,a}] + \textbf{EnergyDelta}(H[k-1, g-x_{\eta,a}], \eta) < M[k-1, g]))$
         then
9:          $M[k,g] = M[k-1,g]$; $H[k,g] = H[k-1,g]$;
10:      else
11:         $M[k,g] = M[k-1, g-x_{\eta,a}] + \textbf{EnergyDelta}(H[k-1, g-x_{\eta,a}], \eta)$;
12:         $H[k,g] = H[k-1, g-x_{\eta,a}]$;
13:         $H[k,g][a] = H[k,g][a] - x_{\eta,a}$; $H[k,g][b] = H[k,g][b] + x_{\eta,b}$;
14:      end if
15:   end for
16: end for
17: $\mathcal{A} = 0$;
18: for $g = 0 \ldots X_a$ do
19:    $\mathcal{A} = \max(\mathcal{A},\ M[Z,g])$;
20: end for
21: RETURN $\mathcal{A}$;
```

4.2 The DP-Based Algorithm

Algorithm 5 implements the recursive formula described in Section 4.1 to determine the group of tasks that should be migrated out of the most-loaded processor C_a for the maximum reduction on $\mathcal{E}$. First, we initialize $M[0,g]$ and $H[0,g]$ for $g = 0$ to X_a (line 2 to 4). We next construct the rest M and H matrices in a dynamic-programming manner. The values of $M[k,g]$ and $H[k,g]$ depend on whether $\beta_{a,k}$ is selected for migration, as shown in Eq. (8). If it is not selected, we simply make a copy of $M[k-1,g]$ and $H[k-1,g]$ (line 9). Otherwise, we update them by including the task of $\beta_{a,k}$ in the migration list (line 11 to 13). Finally, we determine $\mathcal{A}$ by the maximum of $M[Z,g]$ for any possible g (line 18 to 20). The group of tasks that result in $\mathcal{A}$ presents an optimal solution for maximizing the energy reduction under the migration order of β_a.

Each invocation of **MaxReduction** reduces the workload of the most-loaded processor but also increases the workload of other processors. When the time complexity is not a concern, we can keep applying **MaxReduction** on any loaded-processor until no further energy reduction is available or below a certain threshold. We call it the **Fully-Balanced** algorithm that may require an unbounded number of calls on **MaxReduction**. To bound its complexity, we propose another algorithm called **DP-Based** that applies **MaxReduction** on each processor only once, starting with the most-loaded processor C_a. After C_a is done with its energy reduction, **DP-Based** removes it from its selection list and continues to the next most-loaded processor. Let X denote the maximum possible load of a processor,

$$X = \max\{\sum_{i=1}^{n} x_{i,1}, \sum_{i=1}^{n} x_{i,2}, \ldots, \sum_{i=1}^{n} x_{i,m}\}.$$

We bound the complexity of **MaxReduction** by $O(nX)$ and **DP-Based** by $O(nmX)$.

Table 3 shows an example of 3 tasks on a 2-processor system. The **kX^3-Partition** algorithm initially assigns all three tasks to C_1 and takes 2.5 mJ to finish all tasks by its deadline $D = 0.01$ second. We apply **MaxReduction** on C_1 to migrate tasks to C_2. **MaxReduction** constructs the M and H matrices of C_1 as shown in Figures 2 and 3. The values listed in Figure 2 are in the units of 10^{-4}. The largest entry in M is $M[3,4]$, which is the maximum value of $M[2,4]$ and $M[2,3] + \textbf{EnergyDelta}(H[2,3], \tau_3)$. By migrating τ_2 and τ_3 to C_2, we minimize the total energy consumption to 1.18 mJ. The speeds of C_1 and C_2 are set at 300 Hz and 400 Hz, respectively.

	k_1		k_2		D
mW/Hz^3	2×10^{-6}		1×10^{-6}		0.01(s)
	$x_{i,1}$	$F_{i,1}$	$x_{i,2}$	$F_{i,2}$	$min(F_{i,j})$
τ_1	**3**	5.4×10^{-5}	5	1.25×10^{-4}	5.4×10^{-5}
τ_2	**1**	2×10^{-6}	2	8×10^{-6}	2×10^{-6}
τ_3	**1**	2×10^{-6}	2	8×10^{-6}	2×10^{-6}
$total$	5	2.5×10^{-4}	0	0	2.5 (mJ)
	500Hz	250mW	0Hz	0mW	

Table 3: A 3-task 2-processor task set

$g \backslash k$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	1.09	1.09	1.09
2	0	1.14	1.14	1.14	1.09	1.09
3	0	1.14	1.32	1.32	1.32	1.09

Figure 2: The M matrix of C_1

Processor	Min k(mW/Hz^3)	Max k(mW/Hz^3)
ARM92x	1.5026×10^{-5}	3.1855×10^{-5}
ARM10x	3.0469×10^{-6}	3.4466×10^{-6}
ARM11x	4.0718×10^{-7}	1.1478×10^{-6}
TMS320Cx	3.2277×10^{-9}	5.2083×10^{-7}
TMS320Dx	1.1250×10^{-8}	3.5095×10^{-8}

Table 4: The k range in each processor model

task number	6	8	10	12	14	16
2 processors	3	3	3	3	3	3
4 processors	5	5	5	6	6	5
6 processors	7	8	9	8	9	11
8 processors	10	11	12	14	13	12

Table 5: **MaxReduction** calls by `FB`

g \ k	0	1	2	3	4	5
0	X_1=5 X_2=0	X_1=5 X_2=0	X_1=5 X_2=0	X_1=5 X_2=0	X_1=5 X_2=0	X_1=5 X_2=0
1	X_1=5 X_2=0	X_1=5 X_2=0	X_1=5 X_2=0	X_1=2 X_2=5	X_1=2 X_2=5	X_1=2 X_2=5
2	X_1=5 X_2=0	X_1=4 X_2=2	X_1=4 X_2=2	X_1=4 X_2=2	X_1=2 X_2=5	X_1=2 X_2=5
3	X_1=5 X_2=0	X_1=4 X_2=2	X_1=3 X_2=4	X_1=3 X_2=4	X_1=3 X_2=4	X_1=2 X_2=5

Figure 3: The H matrix of C_1

5 Experimental Results

We conducted a series of simulations to demonstrate the effectiveness of our algorithms in delivering the optimal energy-saving performance. There are nearly 30 processors modeled in our simulations. These processors include general-purpose embedded processors, such as ARM9, ARM10, and ARM11, and DSP processors, such as TMS320C and TMS320D. The adjusted switched capacitance of each processor is obtained from the official web site of ARM and TI and is summarized in Table 4. We conducted our simulations on an Intel Xeon server with 1GB memory. We evaluate our algorithms on a simulated HeMP system consisting of 2, 4, 6, and 8 processors, each of which is randomly selected from the list of modeled processors. We vary the workload by changing the number of tasks and each task has an execution cycle count between 1,000 and 3,000. For each configuration, we ran simulations for 30 times and took the average value for comparison.

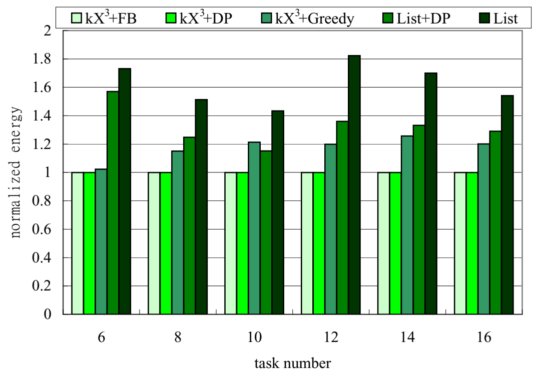
We use `kX`^{3} to denote the **kX^3-Partition** algorithm. We use `List` to denote a commonly-used HoMP low-power scheduling algorithm that dispatches a task to an least-loaded processor [11, 5]. There are 3 energy-reduction algorithms: `Greedy` denotes the **Greedy-Based** algorithm, `DP` denotes the **DP-Based** algorithm, and `FB` denotes the **Fully-Balanced** algorithm. All results are compared to the optimal value that is obtained by exhaustively iterating through all possible task-to-processor assignments and finding the minimum energy consumption. Because the number of iterations grows exponentially with the number of tasks and processors, we cannot obtain the optimal value at the configurations of 16 tasks on 6 processors and 12 or more tasks on 8 processors. For example, the configuration of 16 tasks and 8 processors will take approximately 10 years to finish all iterations. Instead, we use a method of linear regression to obtain these impossible values.

Figure 4 shows the experimental results. Because heterogeneous performance among different processors is not considered, `List` delivers the worst performance in all configurations. At the configuration of 6 tasks on 8 processors, its energy consumption is 30 times of the optimal value. The combination of (`List` + `DP`) significantly reduces its energy consumption. In comparison, the combination of (`kX`^{3} + `DP`) requires considerably less energy than the previous combination. This result well demonstrates the importance of initial task assignments and the effectiveness of our local-optimal partition.

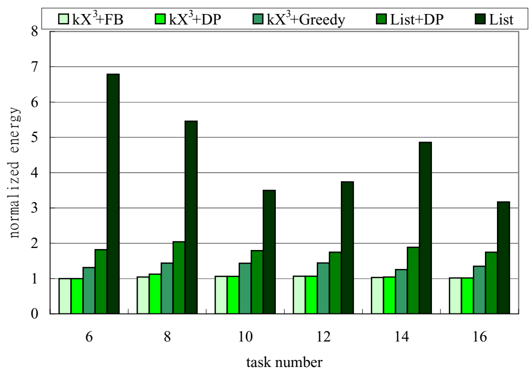
(`kX`^{3} + `DP`) delivers better performance than (`kX`^{3} + `Greedy`) because `DP` consider all tasks while `Greedy` considers only one task for each migration. The difference between (`kX`^{3} + `DP`) and (`kX`^{3} + `FB`) is at their number of **MaxReduction** calls. `DP` calls **MaxReduction** on each processor once while `FB` iteratively calls **MaxReduction** on any most-loaded processor until no reduction is made. Table 5 shows the number of calls by (`kX`^{3} + `FB`). Both combinations deliver almost identical performance at all configurations even though (`kX`^{3} + `DP`) requires less calls on **MaxReduction**. Finally, both combinations deliver the near-optimal energy-saving performance at most configurations. The only exceptions are at the configurations where we can only obtain *approximate* optimal values through linear regressions. These experimental results show that, at its polynomial-time complexity, (`kX`^{3} + `DP`) still yields the near-optimal result for the HeMP single-level voltage setup problem.

6 Conclusions and Future Work

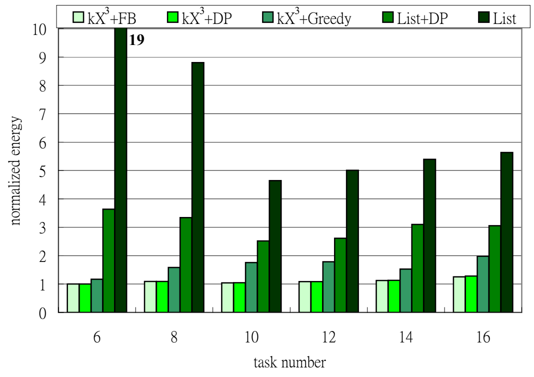
Heterogeneous multi-processor (HeMP) systems are adopted by low-power embedded systems to host different categories of applications. A real-time scheduling algorithm is required to minimize the total energy consumption and complete all tasks before their deadline. In this paper,



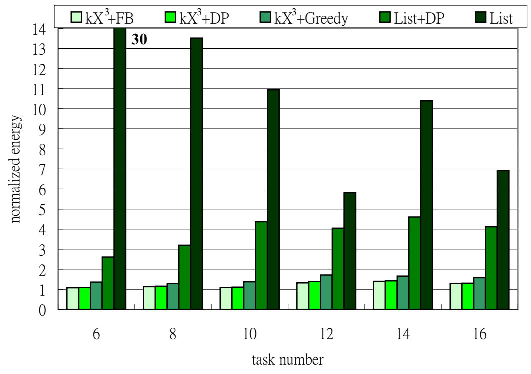
(a) two processors



(b) four processors



(c) six processors



(d) eight processors

Figure 4: The total energy consumption being normalized to the optimal value

we provide a near-optimal solution for the HeMP single-level voltage setup problem in which we determine an optimal speed for each processor to achieve this goal. Our workload consists of a set of frame-based tasks, each of which is independent and non-preemptible. Initially, each task is assigned to a processor in a local-optimal manner. We next provide a couple of polynomial-time solutions to reduce energy. The first solution is based on a greedy-based algorithm to consider one task at a time. The second solution is based on a dynamic-programming algorithm to consider all tasks of a processor during one migration. Finally, we determine a processor's speed by its final workload and the common deadline.

A series of simulations were conducted to demonstrate the effectiveness of our algorithms. We modeled more than a couple dozens of off-the-shelf embedded processors including ARM and TI DSP processors. We compared our algorithms with a commonly-used homogeneous multi-processor (HoMP) scheduling algorithm and the optimal solution. The optimal solution is implemented as an exhaustive iteration of all possible task-to-processor assignments. Using the same energy-reduction method, the local-optimal partition consumes significantly less energy than the HoMP scheduling algorithm. Compared to the optimal solution, the combination of the local-optimal partition and the dynamic-programming method delivers almost identical performance at all measurable configurations. The experimental results demonstrate that our work succeeds to provide a near-optimal solution for the HeMP single-level voltage setup problem at its polynomial-time complexity.

For the voltage setup problem, existing work focused on a one-processor system and has provided a couple of solutions for both the single-level and the multi-level problems. Our work is the first one that addresses the multi-processor voltage setup problem. We started with the HeMP single-level problem in this paper. Currently, we are extending our discussion to solve the HeMP multi-level problem. The HoMP voltage setup problem could be eventually solved in a similar way.

7 Acknowledgment

This work was supported in part by Ministry of Economic Affairs, R.O.C., under Grant MOEA 95-EC-17-A-04-S1-044 and under Grant MOEA 95-EC-17-A-01-S1-038. It was also support by National Science Council (NSC) under contract NSC 95-2221-E-007-049-MY3.

References

[1] J. H. Anderson and S. K. Baruah. Energy-efficient synthesis of periodic task systems upon identical multiprocessor platforms. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems*, pages 428–435, 2004.

[2] J. H. Anderson and A. Srinivasan. Early-release fair scheduling. In *The 12th Euromicro Conference on Real-Time Systems*, pages 35–43, June 2000.

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

[4] J.-J. Chen and T.-W. Kuo. Multiprocessor energy-efficient scheduling for real-time tasks with different power characteristics. In *ICPP '05: Proceedings of the 2005 International Conference on Parallel Processing*, pages 13–20, 2005.

[5] A. Gerasoulis and T. Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):686–701, 1993.

[6] J.-J. Han and Q.-H. Li. Dynamic power-aware scheduling algorithms for real-time task sets with fault-tolerance in parallel and distributed computing environment. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium*, 2005.

[7] H.-R. Hsu, J.-J. Chen, and T.-W. Kuo. Multiprocessor synthesis for periodic hard real-time tasks under a given energy constraint. In *DATE '06: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 1061–1066, 2006.

[8] S. Hua and G. Qu. Voltage setup problem for embedded systems with multiple voltages. *IEEE Transactions on Very Large Scale Integration Systems*, 13(7):869–872, 2005.

[9] C.-H. Lee and K. G. Shin. On-line dynamic voltage scaling for hard real-time systems using the edf algorithm. In *Proceedings of the 25th IEEE Real-Time Systems Symposium*, December 2004.

[10] F. Liberato, S. Lauzac, R. Melhem, and D. Mosse. Fault tolerant real-time global scheduling on multiprocessors. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pages 252–259, 1999.

[11] C. McCreary, A. A. Khan, J. J. Thompson, and M. E. McArdle. A comparison of heuristics for scheduling dags on multiprocessors. In *Proceedings of the 8th International Symposium on Parallel Processing*, pages 446–451, 1994.

[12] O. Paker, J. Sparsal, N. Haandbzk, M. Isage, and M. Isage. A low-power heterogeneous multiprocessor architecture for audio signal processing. *Journal of VLSI Signal Processing System*, 37(1):95–110, 2004.

[13] J. Seo and N. D. Dutt. A generalized technique for energy-efficient operating voltage set-up in dynamic voltage scaled processors. In *ASP-DAC '05: Proceedings of the 2005 Conference on Asia South Pacific Design Automation*, pages 836–841, 2005.

[14] J.-H. Sohn, J.-H. Woo, J. Yoo, and H.-J. Yoo. Design and test of fixed-point multimedia co-processor for mobile applications. In *DATE '06: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 249–253, 2006.

[15] M. T. Strik, A. H. Timmer, J. L. Meerbergen, and G.-J. Rootselaar. Heterogeneous multiprocessor for the management of real-time video and graphics streams. *IEEE Journal of Solid-State Circuit*, 35(11):1722–1731, 2000.

[16] Y. Yu and V. K. Prasanna. Resource allocation for independent real-time tasks in heterogeneous systems for energy minimization. *Journal of Information Sciece and Engineering*, 19(3):433–449, 2003.