POWER-AWARE ONLINE DYNAMIC SCHEDULING FOR MULTIPLE FEASIBLE INTERVALS[*]

Bo Liu, Fang Liu, Jian Lin and Albert MK Cheng

Department of Computer Science
University of Houston
Houston, TX, 77204, USA
`http://www.cs.uh.edu`

Technical Report Number UH-CS-10-06

July 23, 2010

Keywords: Power-aware Scheduling, Multiple Feasible Intervals, Online Scheduling, Dynamic Frequency Scheduling, Real-time System.

Abstract

Multiple Feasible Intervals (MFI) is a task model which has more than one interval for each task instance to run. In our previous work, different scheduling algorithms are proposed for both static MFI dynamic MFI during run-time. Real-life situations demand dynamic MFI, but calculating an MFI schedule is a NP-Hard problem and efficiency of algorithm itself is one of the main concerns. In this paper, we introduce a more efficient online dynamic algorithm, Online Dynamic Multiple Feasible Intervals (ODMFI) which aggressively reduces power consumptions with help from accurate performance prediction models and integration of a simple caching policy into scheduling algorithm to cope with changing feasible intervals during run-time. Our simulations show that ODMFI generates schedules which have close level of energy consumption to Power-Aware MFI [3] with less overhead compared with Dynamic MFI [6].

[*] This work was supported in part by the National Science Foundation under Award No. 0720856.

POWER-AWARE ONLINE DYNAMIC SCHEDULING FOR MULTIPLE SCHEDULING FOR MULTIPLE FEASIBLE INTERVALS[†]

Bo Liu, Fang Liu, Jian Lin and Albert MK Cheng

Abstract

Multiple Feasible Intervals (MFI) is a task model which has more than one interval for each task instance to run. In our previous work, different scheduling algorithms are proposed for both static MFI dynamic MFI during run-time. Real-life situations demand dynamic MFI, but calculating an MFI schedule is a NP-Hard problem and efficiency of algorithm itself is one of the main concerns. In this paper, we introduce a more efficient online dynamic algorithm, Online Dynamic Multiple Feasible Intervals (ODMFI) which aggressively reduces power consumptions with help from accurate performance prediction models and integration of a simple caching policy into scheduling algorithm to cope with changing feasible intervals during run-time. Our simulations show that ODMFI generates schedules which have close level of energy consumption to Power-Aware MFI [3] with less overhead compared with Dynamic MFI [6].

Index Terms

Word Power-aware Scheduling, Multiple Feasible Intervals, Online Scheduling, Dynamic Frequency Scheduling, Real-time System.

I. Introduction

Real-time (RT) systems strive to schedule all the tasks within their respective deadlines and power-aware RT system aims to minimize system's power consumption. Multiple Feasible Intervals (MFI) is a new type of task model that was pioneered in [1, 2] and we targets on power-aware scheduling for MFI.

An MFI task executes during one or more feasible time intervals before its deadline. The Power-Aware MFI algorithm [3], a combination of offline and online scheduling algorithm, uses fetch-ahead and push-back techniques to reduce power consumption. It calculates schedules based on known information which includes available intervals. However, task attributes, such as the available intervals, often change during run-time with the fluctuations of various circumstances. Dynamic Multiple Feasible Intervals (DMFI) [6] is an online scheduling algorithm proposed to handle the above situations. It mainly solves two problems found in our previous online scheduling: low utilization of un-passable slacks, and low efficiency of algorithm itself.

The rest of the paper is organized as follows. Section 2 explains the limits of our previous work, the Power-Aware MFI scheduling and DMFI scheduling. Section 3 introduces the fetch-ahead and push-back algorithms which are building blocks in our algorithm. Section 4 introduces ODMFI. In details, Section 4.1 defines ODMFI system; Section 4.2 proposes ODMFI algorithm and explains its advantages in efficiency and in reducing un-passable slacks, i.e. scaling CPU frequency more aggressively. Section 4.4 illustrates how it works. And Section 5 states our experiments and explains their results. Finally, Section 6 concludes our paper.

II. Power-aware MFI and DMFI

Our previous work provides two foundation elements for ODMFI. Besides saving the schedulability and energy efficiency, Power-Aware MFI scheduling is a static feasible algorithm, while DMFI is a dynamic feasible algorithm which has heavy overhead itself. In this paper, we propose an efficient algorithm ODMFI to handle the following situation which DMFI addresses previously.

[†] This work was supported in part by the National Science Foundation under Award No. 0720856.

Most of time, the intervals may not be known beforehand and the intervals may be changed during run-time. In addition, determining the intervals in the first place often depends on outside factors not known until run-time or even until the task has already started.

Consider this example. Country A launches a cruise missile at Country B. Country B wants to jam that cruise missile without the missile falling onto a population center; this is an MFI problem. Using the original fetch-ahead and push-back algorithms, this task could not be accomplished unless Country A notified Country B beforehand and the two countries plotted the trajectory of the missile beforehand. In reality, that will never happen.

The current Power-Aware MFI algorithm is not capable of coping with such a situation because it assumes the intervals are static and known a priori. Hall et cl. proposed the above problem in [6] and attempts to solve it with a piece-wise online version of Power-Aware MFI scheduling, Dynamic MFI (DMFI) [6]. Since DMFI is a modified Power-Aware MFI algorithm, it inherits one problem: low utilization of un-passable slacks [3]. This is one of the problems that we propose Online Dynamic MFI (ODMFI) to solve in this paper. Section 4 explains how ODMFI utilizes performance model to aggressively scale CPU frequency. The other problem is algorithm overhead. Calculating MFI schedules is a NP-Hard problem and Power-Aware MFI takes heavy calculations offline to improve algorithm efficiency. Because DMFI does all the calculation on the fly, it has much higher cost than Power-Aware MFI. Section 4 also explains our solution in details.

A. *Power-aware MFI Scheduling*

As mentioned above, Power-Aware MFI scheduling is a combination of static and dynamic schema. The static schema fully utilizes the known attributes of task sets, except the actual execution time, and runs an offline scheduling. The dynamic schema takes the calculated schedule and applies a greedy algorithm to take advantage of available slacks (definition in Section 4.1). It was not designed to handle the dynamic situation as the above, because the intervals may change after tasks are ready. DMFI scheduling is the first designed to handle situations like interval changes proposed.

B. *Subsection B*

DMFI scheduling is modified based on Power-Aware MFI algorithm to cope with the above situation. It creates a small piece of the Power-Aware MFI schedule at a time. The algorithm includes embedded loops in $recalculate\_intervals()$ which rely on two parameters, α and β in the function as described in [3]. The α and β rely on the attributes of the task sets which are known during run-time and thus β is determined through multiple tries during run-time as well. Since DMFI repeats its search in a foot step of β, the worst computation time of DMFI is N^2 when N tasks are within β time units, where N is the number of the tasks in the task set. Thus its complexity is $O(N^2)$.

III. MFI Push-back and Fetch-Ahead Algorithms

MFI push-back algorithm is a base element of our online dynamic algorithm. Since the future tasks cannot be predicted, the push-back algorithm copes with the ready tasks. The push-back algorithm locates slacks left by completed tasks. A task is rescheduled in this slack only if its interval and the slack have an intersection and the intersection is larger than its execution time. Thus, it saves power due to CPU frequency scaling as explained in Section 4.2.

Figure 1 illustrates the schedule of the following task set by push-back algorithm. Task 2 starts at time unit 8 and finishes at time unit 10. Since task 4 cannot be scheduled in its earlier feasible intervals, it is rescheduled to use the interval (10, 15] where has enough time units to finish task 4 and save the power consumption. Task 5 is scheduled on time unit 15 and finishes on time. The overall schedule saves power consumption due to longer execution time of task 4.

Task	Feasible Intervals	Comp. Time
T1	(1,9]	2
T2	(2,4], (8,13]	2
T3	(1,4], (5,8], (9,13]	2
T4	(1,8], (4,9], (10, 15]	3
T5	(1,7], (15,18]	3

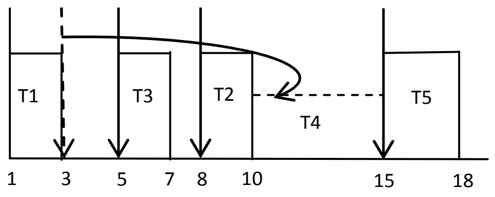

Fig. 1. Result of push-back algorithm: Push-back algorithm schedules Task 4 into the slack between Task 2 and Task 5.

MFI fetch-ahead algorithm is similar with MFI push-back algorithm. Instead of looking for an available slack after the following tasks, the fetch-ahead fits the following tasks in a slack somewhere in front of their release times. Regarding energy consumption, push-back and fetch-ahead algorithms have the same effects which takes advantage of the slacks to scale the task execution time as much as possible.

IV. Online Dynamic MFI

We solve the problem described in section 2 with Online Dynamic Multiple Feasible Intervals (ODMFI) algorithm. ODMFI schedules an MFI task set with the following changes of the task properties:

- The release times are not known until run-time;
- The intervals are not known until run-time; and
- The intervals can change during run-time.

As stated before, the MFIs of a task are often dependent on outside conditions that are not known until the task set is already running. ODMFI is designed as an online algorithm to schedule such MFI task sets with unknown intervals before run-time and intervals that change during run-time. Thus, the ODMFI algorithm copes with scenarios like the missile problem.

A. ODMFI Problem Definition

ODMFI is an attempt to simulate a real-life MFI schedule. In a real-life situation, the tasks do not begin as an MFI task. The tasks begin as EDF [7] tasks and by considering outside factors, current circumstances, and other tasks, multiple feasible intervals are determined for each task. The introduction of new outside factors, circumstances, and tasks can change the feasible intervals of the current tasks, and ODMFI attempts to accommodate that possibility.

Since the ODMFI is an online dynamic version of the MFI algorithms previously discussed in Section 2, it makes many of the same assumptions. The ODMFI algorithm uses the following system model.

The tasks in the system are EDF-schedulable tasks. As in [12], we don't assume that the task must be periodic. Tasks are supposed to be independent and have arbitrary release time and deadline. A task instance in a periodic task is thus considered as an independent task in our system. A task T_i possesses the following parameters [7]:

- Multiple intervals $I_{i,j} = (S_{i,j}, E_{i,j}]$ where $S_{i,j}$ is the start point of interval $I_{i,j}$ and $E_{i,j}$ is the end point of interval $I_{i,j}$, $0 < j \leq M_i$ where M_i is the total number of T_i's intervals, and these intervals are mutually exclusive;
- Release time r_i is the time at which a task becomes ready for execution;
- Start time s_i is the time at which a task starts its execution;
- Finishing time f_i is the time at which a task finishes its execution;
- Execution time e_i is the difference between the finishing time f_i and the start time s_i, $e_i = f_i - s_i$;
- Response time R_i is the difference between the finishing time f_i and the release time r_i, $R_i = f_i - r_i$;
- Worst case execution time (WCET) C_i is the maximum time task T_i needs to finish its execution under all circumstances;
- Absolute deadline d_i is the time before which task T_i should be completed to avoid damage, and in our system $max(E_{i,j}) \leq d_i$; and

- Slack time X_i is the maximum task T_i can be delayed on its activation to complete within its deadline, $X_i = d_i - r_i - e_i$.

MFI task model assumes that these tasks can only run during certain time intervals before their respective deadlines. The tasks' intervals $I_{i,j}$ are known after they are released and then they become MFI tasks. Naturally, each task's intervals are determined along the time line and the intervals must all end by its deadline.

When a new task is released, the intervals of one or more tasks, including the executing intervals, may be changed. This is due to the fact that the time intervals of a task may be dependent on other tasks and outside factors that are not known until run-time. In the cruise missile example, if the missile changes its target mid-flight, the trajectory of the missile changes. Since the missile will be flying over different population centers, the time intervals to jam the missile change as well.

Without losing generality, we can assume the outside factors only add new feasible intervals in the future after run-time changes. It's equivalent to extend the end times of current feasible intervals which allow the tasks execute in the extended time period in the current intervals. For example, we suppose $\left(S_{i,j}, E_{i,j}\right]$ is one of the execution intervals to jam a missile. When the missile change its target mid-flight at a certain time point t, we can either add a new feasible interval $\left(S_{i,j+1}, E_{i,j+1}\right]$ where $S_{i,j+1} \geq t$, or extend $\left(S_{i,j}, E_{i,j}\right]$ to $\left(S_{i,j}, E'_{i,j}\right]$ where $E_{i,j} < t < E'_{i,j}$. Since the target change is known after t, the task must execute in $\left(t, E'_{i,j+1}\right]$ which has the same effect as adding a new feasible interval after t. In our experiments of the ODMFI algorithm in Section 5, the tasks' intervals $\left(S_{i,j}, E_{i,j}\right]$, t and interval changes are simulated with random numbers which are sorted to simulate the fact that they are determined along the time line after r_i.

B. *ODMFI Algorithm*

As other power-aware algorithms, the design of ODMFI algorithm has two goals. The first is to reduce energy consumption and the second goal is to minimize missed deadlines. As mentioned in Section 3, it reduces energy consumption by utilizing slacks to scale tasks. As in the previous work, we follow the power function $Power(CPU) \propto (CPU)^2$ [9] where CPU represents CPU frequency, and the scale factor sf is calculated as $sf = execution\_time_{current}/execution\_time_{new}$ where $execution\_time_{current}$ is the execution time with default CPU frequency set by operating system and $execution\_time_{new}$ is the new execution time after the CPU frequency is reset to $CPU = default\_CPU \cdot sf$ where $default\_CPU$ is CPU frequency set by default. Based on the power function, to save energy, it's necessary to have $sf < 1 \Rightarrow execution\_time_{new} > execution\_time_{current}$ to save energy. From our system model, we only know one execution time before the task starts to run: WCET. We initialize our $sf = C_i/I_i$ where C_i is the WCET for task T_i and I_i is its execution interval. It's well known that WCET is conservative and we cannot scale CPU frequency small enough to run the task as long as its execution interval. To reduce power consumption aggressively, the actual execution time e_i is the ideal candidate to calculate scale factor. To get the closest estimation of e_i, we build a prediction model for each task based on its execution time records and update the scale factors with predicted task execution time ee_i. Section 4.3 describes how we build the prediction models.

The difficulty for ODMFI is to schedule online MFI tasks with changing information. To dynamically utilize fetch-ahead and push-back online, we cache a certain number of tasks before generating schedules instead of looking at every task as in offline scheduling of Power-Aware MFI. Since the feasible intervals are available after tasks are released, we run our ODMFI before the tasks start to execute. That is, it requires that our ODMFI finds a schedule during period $s_i - r_i$. We have two steps to finish here: 1) cache released tasks; and 2) run ODMFI. It takes our slack time for these two steps to run. Thus, it's necessary to design and implement ODMFI very efficiently.

The details of ODMFI are in Figure 2 and Figure 3. Symbol // starts a line of comments. It first puts all the released tasks in a queue $Queue\_T(N)$ which has a capacity of N tasks. $Queue\_T()$ is a First In First Out (FIFO) queue which maintains tasks' release order. Each task joins the queue with multiple intervals which are listed in an end time, $E_{i,j}$, non-decreasing order like how the feasible intervals listed in Figure 1 example.

The algorithm maintains a queue $Queue\_IE(M \cdot N)$ which contains the ordered end times of feasible intervals of ready tasks. $Queue\_IE()$ is a queue of interval end points in which $min\left(I_{i,j}\right)$, $1 \leq i \leq N$, $1 \leq j \leq max(M_i)$, is the first node in the queue. The capacity of $Queue\_IE(M \cdot N)$ is $M \cdot N$ where M is the maximum number of feasible intervals a task can have, $M = \max_{0 \le i \le N} M_i$, and N is the maximum number of tasks that $Queue\_IE()$ can handle. $Queue\_IE()$ is empty first. The sorted feasible intervals of the ready tasks are merged into the queue one by one after they arrive. If the two intervals have the same end times, the start time is used to decide their orders. For example, T_1 has a feasible interval $I_{1,1} = (1, 9]$; T_2 has feasible intervals $I_{2,1} = (2, 4]$, $I_{2,2} = (8, 13]$; T_3 has feasible intervals $I_{3,1} = (1, 4]$, $I_{3,2} = (5, 8]$, $I_{3,3} = (9, 13]$. $Queue\_IE()$ first has $I_{1,1} = (1, 9]$ after T_1 arrives, then $I_{2,1}$ and $I_{2,2}$ are merged into the queue after T_2 arrives and results into a queue as $I_{2,1} = (2, 4]$, $I_{1,1} = (1, 9]$, $I_{2,2} = (8, 13]$; then a new queue is generated as $I_{3,1} = (1, 4]$, $I_{2,1} = (2, 4]$, $I_{3,2} = (5, 8]$, $I_{1,1} = (1, 9]$, $I_{2,2} = (8, 13]$, $I_{3,3} = (9, 13]$ after T_3 arrives. Tasks execute based on ordered $Queue\_IE()$.

The other queue $Queue\_IS(M \cdot N)$ is used in $imperfect\_fetch\_ahead()$. It's built in a similar process with a non-decreasing order of start times, and the end time is used for break-even in the same way. It's used to look for intervals which have late end times $E_{i,j}$ but early start time $S_{i,j}$ like interval $I_{1,1}$. Such intervals are inserted in the back of $Queue\_IE()$ due to their large end time values, and in the front of $Queue\_IS()$ due to their small start time values. Let's compare task T_1 and task T_2. Because $E_{1,1} > E_{2,1}$, the algorithm will pick up task T_2 earlier than task T_1 from $Queue\_IE()$. However, due to $S_{1,1} < S_{2,1}$, task T_1 has a chance to be picked up by $imperfect\_fetch\_ahead()$ to fit in a slack left by a task run earlier.

Our ODMFI algorithm preserves EDF nature of MFI task sets. The reason is that $Queue\_IE()$ is sorted with non-decreasing order of interval end times and $max(E_{i,j}) \le d_i$, which means task instances execute with Earliest Deadline (End time) First. ODMFI effectively utilizes MFI fetch-ahead and push-back algorithms mentioned in Section 3. The push-back algorithm is rooted in the design due to the fact that ODMFI scheduling mainly follows $Queue\_IE()$ which maintains EDF in the design and thus early released tasks may run late. $imperfect\_fetch\_ahead()$ makes Fetch-ahead possible. It's illustrated in Figure 5 in Section 4.3: Task 4 is not released until interval (10,15] although task 4 is ready at time unit 1.

```
Model {
    Text Component;
    Model Pointer model_pointer;
};
Interval {
    Long Integer start_time;
    Long Integer end_time;
    Task Pointer task_pointer;
};
Task {
    Integer index;
    Model Pointer model_pointer;
    Long Integer release_time;
    Long Integer start_time;
    Long Integer deadline;
    Long Integer WCET;
    Long Integer predicted_execution_time;
    Long Integer actual_execution_time;
    Interval $I_1$;
    … …
    Interval $I_{M_1}$;
    Instance job;
};
```

Figure. 2 Data structures of ODMFI algorithm

Due to the NP-hardness of the problem [10], we design a low-cost ODMFI by: 1) limiting number of steps in our search to some constants instead of exhausting search as all the heuristic algorithms, including caching n tasks in $Queue\_T()$ and only partially searching $Queue\_IS()$ within a constant steps p; and 2) duplication in data structure defined as in Figure 1. Each task data structure has M_i pointers linking to its task. We can reach a certain task from its corresponding node in either $Queue\_IE()$ or $Queue\_IS()$ through its task pointer instead of searching $Queue\_T()$. Function $Imperfect\_fetch\_ahead()$ stops after checking p intervals in $Queue\_IS()$. Parameter p

obviously affects the performance of ODMFI which adds $O(p)$ to each loop and the overall response time of the ODMFI algorithm.

```
ODMFI Algorithm
1.  Merge $I_{i,j}$ into $Queue\_IE()$ and $Queue\_IS()$
2.  While($Queue\_IE()$ is not empty) {
3.    // check the head node in the queue $I_{i,j}$
4.    If the interval has an empty task link, move head pointer to the next interval until an interval $I_{i,j}$ with non-empty task link
    is found;
5.    // choose scale factor and return True if there is un-passable slacks
6.    If ( Scale_factor() )
7.      // fetch ahead a feasible task
8.      Imperfect_fetch_ahead();
9.      Run the task $T_{i'}$ in head node of $Queue\_IE()$;
10.     Release resources of chosen interval from $Queue\_IE()$, $Queue\_IS()$ and corresponding task node in $Queue\_T()$;
11. };
12.
13. Scale_factor() {
14.   $sf = 1$;
15.   // check $Queue\_IE()$ and $Queue\_IS()$ to find if any tasks in the queue can fit in the slacks
16.   If ( ( start time $S_{p,q}$ of next interval in $Queue\_IE()$ > end time $E_{i,j}$ of current interval in $Queue\_IE()$ )
17.        || ( $E_{i,j} < S_{m,n}$ of head node of $Queue\_IS() < S_{p,q}$ ) ) {
18.     $sf = ee_i / I_{i,j}$;
19.     Set Boolean to True;
20.   }
21.   else{
22.     $sf = ee_i / (current\ time - r_i + ee_i)$;
23.     Set Boolean to False;
24.   }
25.   Return Boolean;
26. };
27. Imperfect_fetch_ahead() {
28.   Initialize Pointer with head node of $Queue\_IS()$;
29.   While( check less than $p$ intervals in $Queue\_IS()$ ) {
30.     // check $Queue\_IE()$ and $Queue\_IS()$ to find if there is a feasible task to fetch ahead
31.     If( ( $E_{i,j}$ of current task's interval in $Queue\_IE() < S_{m,n}$ of Pointer in $Queue\_IS() < S_{p,q}$ of next interval in
    $Queue\_IE()$ )
32.       && ( $ee_m \le (S_{p,q} - S_{m,n})$ ) ) {
33.         Move interval $I_{m,n}$ after $I_{i,j}$;
34.         Break;
35.       }
36.       Move to next interval;
37.   }
38. }
```

Figure 3 Sudo code of ODMFI algorithm

The complexity of ODMFI is $O(p \cdot M \cdot N)$ where M is the same as defined in the system model and N is the total number of tasks in the task set. Merge in step 2 can be implemented with $O(M \cdot N)$ complexity because intervals are ordered when tasks are released. $imprefect\_fetch\_ahead()$ in $While$ loop of step 3 takes $O(p)$ as mentioned above and other lines takes $O(1)$, so the overall complexity of $While$ loop is $O(p \cdot M \cdot N)$. Thus, the overall complexity of ODMFI is $O(M \cdot N) + O(p \cdot M \cdot N)$. When $p \cdot M \ll N$, ODMFI is much more efficient compared with $O(N^2)$ of DMFI. It's obvious that both p and M have impact on algorithm efficiency and the level of energy consumption reduction. Section 5 illustrates it in details with Figures.

C. *Feedback-based Execution Time Prediction*

We use the priori method to determine computation time of tasks to utilize un-passable slacks better. It's difficult to construct an accurate model based on historical data, especially for those providing complex on-demand computation. However, we introduce a simple modelling procedure which can reach accuracy in the range

83%~99.6% (over 90% most of the time) in our experience. We currently use generalized least square (gls) in S-plus [13] to linearly combine computation time of each component in a task. For example, if a task involves two steps, executing an algorithm of complexity $O(n \cdot logn)$ and then executing an algorithm of complexity $O(n^2)$ with n being problem size, the overall computation time of the task $ComputationLoad$ is modelled as a linear combination of $O(n \cdot logn)$ and $O(n^2)$ by gls(). The final model is generated as $C_1 \cdot n^2 + C_2 \cdot nlog(n) + C_3$ where C_1, C_2 and C_3 are constants fitted by S-plus gls() function.

Equation (1)~(5) presents the synthesized formal model of the overall model to estimate task execution time which is parameterized by problem size. k_1 and k_2 are constants fitted by our chosen fitting function. $psize$ is the problem size. $ComputationLoad$ is the total number of clock-cycles the computation takes in the processor without considering memory references. $MissCount$ is the total number of misses in an arbitrary level of memory hierarchy. $MissPenalty$, the penalty for a miss in an arbitrary level of the memory hierarchy is the difference between the access time to the next memory level and the access time to the current memory level, i.e. difference between latencies (expressed as Lat. in equation (5)) to adjacent memory hierarchy levels.

$$EstimatedET(psize) = (A + B + C)/CPU \cdots\cdots (1)$$
$$A = ComputationLoad(psize) \cdots\cdots\cdots\cdots\cdots\cdots\cdots (2)$$
$$B = k_1 MissCount(psize) \times MissPenalty(L_1) \cdots (3)$$
$$C = k_2 MissCount(psize) \times MissPenalty(L_2) \cdots (4)$$
$$MissPenalty(L_i) = Lat.(L_{i+1}) - Lat.(L_i) \cdots\cdots\cdots (5)$$

In the above equations, CPU and $MissPenalty$ are architecture characteristics which can be found from architecture data sheets [5] or benchmarks [14, 15]. And applications are characterized by application specific models such as $ComputationLoad$ and $MissCount$.

These models can be built on readings from Hardware Performance Monitor Counters (HPMC) [17]. We use HPMC readings of different problem sizes to generate both $ComputationLoad$ and $MissCount$ models from gls() in the similar way with the above. Since problem size is one of the factors which determine the size of loaded data into memory hierarchy, we use $psize$ as a parameter to feed gls() curve fitting. For example, if a matrix of $psize \cdot psize$ is accessed in the algorithm, we generate a model from gls() as $C_1 \cdot n^2 + C_2 \cdot n + C_3$. For memory hierarchy, Intel XScale microarchitecture provides the following hardware event units which are used in our model: Instruction Cache Efficiency, Data Cache Efficiency, Instruction TLB Efficiency, Data TLB Efficiency, Stall/Writeback statistics and Data/Bus Request Buffer Full [16, 17].

D. *ODMFI Examples*

To illustrate schedulability of ODMFI and energy reduction, we consider two task sets as described in ODMFI scheduling. Scheduling the same task set used in Section 3 illustrates push-back effect as in Figure 4 and the other one described in the following table illustrates fetch-ahead effect as in Figure 5.

Task	Feasible Intervals	Comp. Time
T1	(1,9]	2
T2	(6,10], (8,13]	2
T3	(1,12], (13,20]	2
T4	(8,11], (13, 15]	3

Regarding the task set in Section 3, $Queue\_IE()$ after merging is $I_{3,1} = (1,4]$, $I_{2,1} = (2,4]$, $I_{5,1} = (1,7]$, $I_{4,1} = (1,8]$, $I_{3,2} = (5,8]$, $I_{1,1} = (1,9]$, $I_{2,2} = (8,13]$, $I_{3,3} = (9,13]$, $I_{4,3} = (10,15]$ and $I_{5,2} = (15,18]$. In Figure 4, T_3 is released first due to its earliest end time and earliest release time. T_5 is scheduled right after T_3 because $I_{2,1}$ is unusable and T_3 is already released. Since T_3 finishes 1 time unit earlier than its corresponding deadline 4, T_5 gets the slack from T_1 and its execution time is scaled to 4 time units. T_1, T_2 and T_4 all arrived earlier than T_5 but are pushed back into later intervals as the following. T_4 is scaled from 3 time units into 4 time units. The total energy used without scaling is 15, while the total energy used after scaling is $2 + 0.75^2 \cdot 4 + 2 + 2 + 0.75^2 \cdot 4 = 10.5$ which saves 30% energy.

Regarding the task set in this section, $Queue\_IE()$ after merging is $I_{1,1} = (1,9]$, $I_{2,1} = (6,10]$, $I_{4,1} = (8,11]$, $I_{3,1} = (1,12]$, $I_{2,2} = (8,13]$, $I_{4,2} = (11,15]$ and $I_{3,2} = (13,20]$. In Figure 5, T_1 is released first due to its earliest end time and earliest release time. T_3 is scheduled right after T_1 because $I_{2,1}$ is unusable and T_3 is already released. Since T_3 finishes 1 time unit earlier than its corresponding deadline 4, T_3 gets the slack from T_1 and its execution time is scaled to 4 time units. T_1, T_2 and T_4 all arrived earlier than T_3 but T_3 is fetched ahead into earlier interval as in Figure 5. T_3 is scaled from 2 time units into 3 time units which sets the scale factor to 0.75. The total energy used without scaling is 9, while the total energy used after scaling is $2 + 0.75^2 \cdot 3 + 2 + 2 = 7.6875$ which saves 14.58% energy.

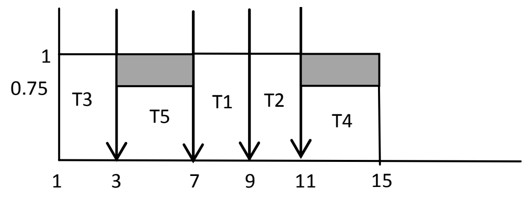

Figure 4. Push-back effect of ODMFI algorithm and energy consumption reduce.

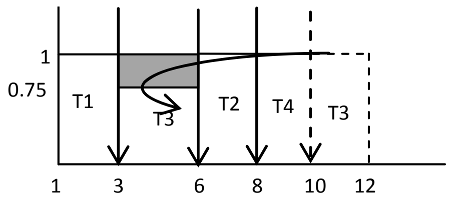

Figure 5. Fetch-ahead effect of ODMFI algorithm and energy consumption reduce.

V. Experiments

In this section, we describe the experiments and the platform, PHYTEC rapid development kit, where they were performed.

A. *Experiment platforms*

PHYTEC rapid development kit includes two parts, the carrier Board and the phyCORE-PXA255 System on Module.

The phyCORE PXA255 [5] module is the core part of the Rapid Development Kits. In order to focus on the interested portions, we have disabled the ports and controllers, such as the CAN controller and USB etc., as well as disconnecting the LCD touch screen. To measure the power consumption, we utilize NI DAQ USB-6008 which provides basic data acquisition functionality for applications such as simple data logging, portable measurements etc. with the provided software package.

The carrier board with the PXA255 computing module plugged is connected to an external stabilized voltage power supply (12V). In order to measure the energy consumed by the computing module, we measure the energy used by the development kit which includes the carrier board and the computing module. To determine the energy consumed by the kit, we connect a small resistor (0.67Ω) in the power supply circuit in series. Since the voltage drop across the resistor is very small and can be safely ignored. The voltage drop across this resistor is measured by a data acquisition, DAQ USB-6800 from National Instruments. The software came with DAQ USB-6800 reads and records the voltage drop 50 times per second. The instantaneous current flowing into the system can be calculated by dividing the voltage drop by the resistance value. Since the voltage of the external supply is stable, the instantaneous power consumption of the development kit can be calculated as $P_{kit} = IV_{external}$. In the equation, I is the recorded current used by the kit, and $V_{external}$ is the external supply's voltage, 12V in our framework. The definite integral of P_{kit} over time interval $[t1, t2]$ is the energy consumption of the kit during the period $E_{kit} = \int_{t1}^{t2} \mathrm{P_{kit}} dt$.

To build execution prediction model based on equation (1)~(5), multiple performance monitoring runs can be done, capturing different events from different modes, because only two events can be monitored at any given time.

For example, the first run could monitor the number of writeback operations (PMN1 of mode, Stall/Writeback) and the second run could monitor the total number of data cache accesses (PMN0 of mode, Data Cache Efficiency). From the results, a percentage of writeback operations to the total number of data accesses can be derived. To make sure these monitoring runs are finished in the same run-time environment and they catch the application characteristics accurately, we ensure that: 1) all performance monitor runs are done exclusively on the environment; 2) each performance monitor run is executed with cold data cache and instruction cache which is enabled by executing other applications in advance. From memory hierarchy performance point of view, 2) is a conservative prediction. We have both periodic and aperiodic tasks, although we schedule them independently. In situation of 2), if instances of a periodic task are scheduled one after one, the above measurement readings are larger than the actual execution times in this schedule. However, it's still more aggressive than using WCET in CPU frequency scaling, which benefits reducing un-passable slacks as indicted in Section 4.2.

B. *Simulations and results*

We use four out of six CPU frequencies in our experiments: 99MHz, 199MHz, 298MHz and 398MHz due to their efficient power consumptions when combined with memory chip frequencies [5, 8]. By default, we use 398MHz which provides shortest execution time for each task on our experiment platform. We normalize CPU frequencies by 398MHz, then we get the following normalized CPU frequency factors $1 = 398MHz/398MHz$, $0.75 = 298MHz/398MHz$, $0.5 = 199MHz/398MHz$, $0.25 = 99MHz/398MHz$. During run time, we choose the closest CPU frequency factor based on the calculated scale factor, e.g. 0.75 is chosen if $C_i/I_{i,j} = 2/3 = 0.67$.

The task set parameters are generated as random numbers, including release times, execution times, intervals and WCETs. Each task set schedule is run 10 times and the average is taken in the final comparison. We mainly compare five algorithms for energy consumption: 1) EDF which has no slack utilization at all; 2) ODMFI_WCET which is ODMFI with conservative scaling of CPU frequency using WCET instead of predicted task execution time in the algorithm; 3) ODMFI_EE which is feedback-based ODMFI with more aggressive scaling of CPU frequency using predicted task execution time ee_i in the algorithm; 4) DMFI which generates small pieces of Power-Aware schedule utilizing more slacks generated with a heavier algorithm; and 5) Power-Aware which fully takes advantage of offline knowledge of the task sets and utilizes the calculated results during online schedule. From the usage of slacks point of view, Power-Aware schedules have the most efficient energy consumption and EDF consumes the most energy due to no slack advantage. It's worth to notice that ODMFI with predicted task execution time scaling still consumes more power than DMFI which actually has a more complete view of the task relationship. Both Figure 6 and Figure 7 show this trend of their energy consumption order corresponding to their slack utilization. The difference between Figure 6 and Figure 7 is that these algorithms behave closer to each other when number of tasks increases, while their energy consumptions spread wider when number of intervals increases. The reason of these trends is because that these algorithms can find less slacks thus less scaling with more tasks, while they can find more slacks thus more scaling room when the number of intervals increases.

We also compare the performance of MFI algorithms, including Power-aware, DMFI and ODMFI in Figure 8. The results are as expected: the performance of the algorithms drops when more ready knowledge is used in the algorithm and thus higher complexity. However, we are coping with changing task attributes, so when the number of intervals increases, M is closer to N and the cost of ODMFI is closer to DMFI. The difference between ODMFI_WCET and ODMFI_EE is the calculation time of task execution prediction models.

In the above experiments, all have $p = 10$ for feedback-based ODMFI, and $\alpha = \beta = 5$ for DMFI. However, as mentioned above, these tuning parameters have impacts on the level of energy consumption and performance of algorithms themselves. In Figure 9 and Figure 10, we show the trend of impacts of p value: as p increases, algorithm cost increases but the energy consumption drops. As noticed in Figure 9, scale factors of CPU frequency are less aggressive in ODMFI_WCET, and thus slacks are larger, which provides more possibilities for fetch-ahead. We notice that energy consumption from $p = 50$ to $p = 100$ is flat, which is because fetch-ahead affects energy consumption on the similar level from $p = 50$ to $p = 100$. It's the same as in the range of $p = 110$ to $p = 200$. However, fetch-ahead effect overall has less impact than that of aggressive scale factors, so ODMFI_WCET has larger energy consumption than feedback-based ODMFI in the whole range. In these experiments, number of tasks is fixed at 200 and number of intervals is fixed at 10.

A critical component of feedback-based ODMFI is the prediction models. An accurate prediction models can benefit both reducing energy consumption and keeping missing deadline ratio as low as possible. We compare its impact on energy consumption with ODMFI with WCET in Figure 6, 7 and 8. In Figure 11, we show the accuracy of our prediction models for two image processing tasks. The curves are fitted based on readings of problem size 48 to size 100 from Hardware Performance Monitor Counters. Our prediction models show over 95% accuracy mainly because they are computation intensive applications and their behaviour follows their algorithm complexity closely. That is, $O(psize^2) + O(psize)$ predicts behaviour of both $ComputationLoad$ and $MissCount$ very well. As mentioned in Section 4.3, our prediction models have accuracy in range of 83%~99.6% with experience of our experiments, which benefits aggressively scaling the CPU frequency and thus aggressively reducing power consumption.

VI. Conclusion

The ODMFI algorithm proved to effectively reduce power consumption compared to our previous work. The ODMFI algorithm is the most computation cost efficient among the three due to the less utilization of offline knowledge and thus it has the most energy consumption. When we integrate our prediction models into ODMFI algorithm, we further reduce tasks power consumption.

There are still some feasible MFI schedule configurations that will cause a task to miss its deadline when using ODMFI and feedback-based ODMFI algorithm. However, ODMFI algorithms improve Power-Aware in the area of calculating an MFI schedule at run-time and improve DMFI in the area of algorithm efficiency. Since the ODMFI algorithms can be implemented with low power and computational costs, then embedded systems such as cell phones and cruise missiles will benefit.

It's also worth to mention the immediate future work to further study the relationship of memory access pattern and power consumption. In ODMFI schedule, although we take memory hierarchy into account, we mainly integrate it into our execution time prediction models, which is based on the power function $Power(CPU) \propto (CPU)^2$. As mentioned by Snowdon et. cl. in [11], the energy consumption patterns of memory chips are heavily dependent on the application's data access behaviour which is different from the power function whose concern is speed or CPU frequency scaling. Due to the complexity of data access patterns of data intensive applications, much study needs to do to understand if ODMFI works well when considering both CPU and memory access patterns.

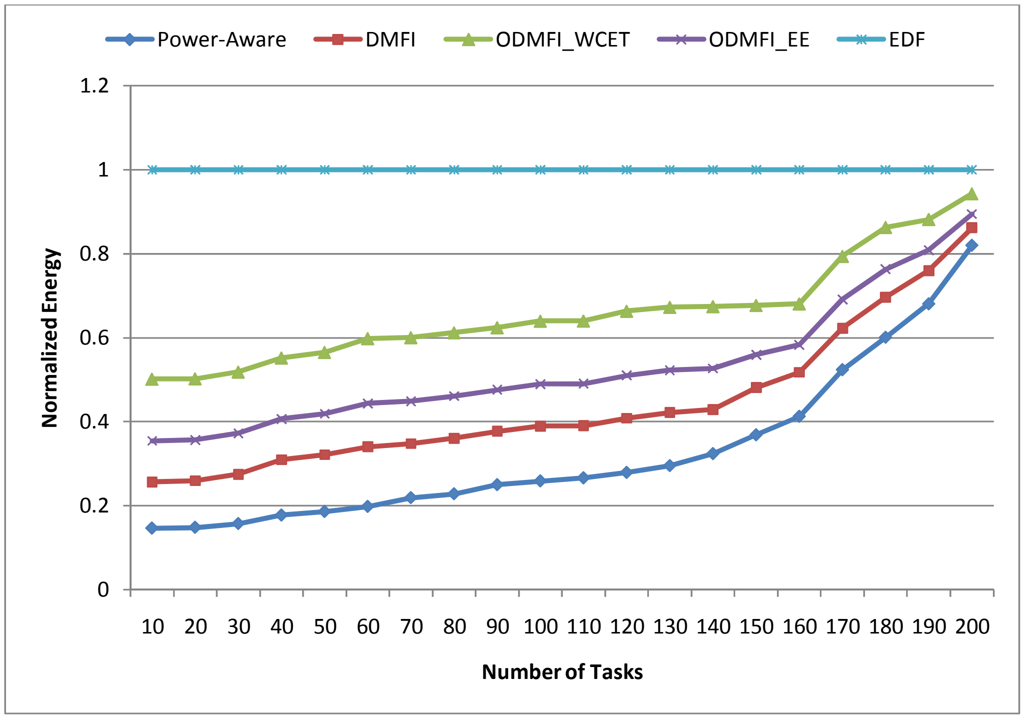

Figure 6. Algorithms comparison based on number of tasks.

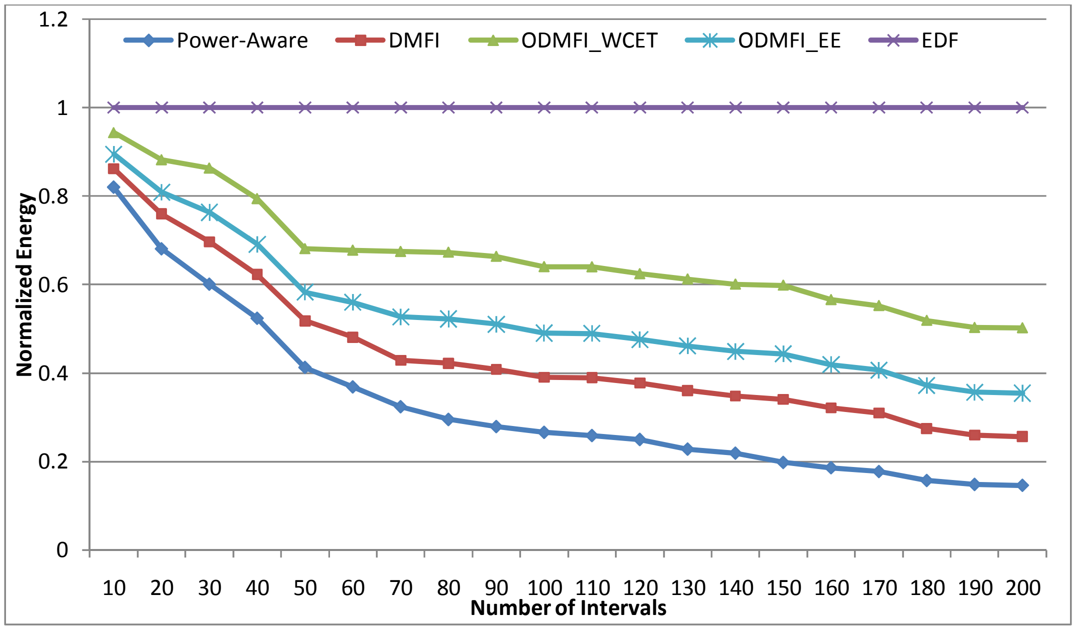

Figure 7. Algorithms comparison based on number of intervals.

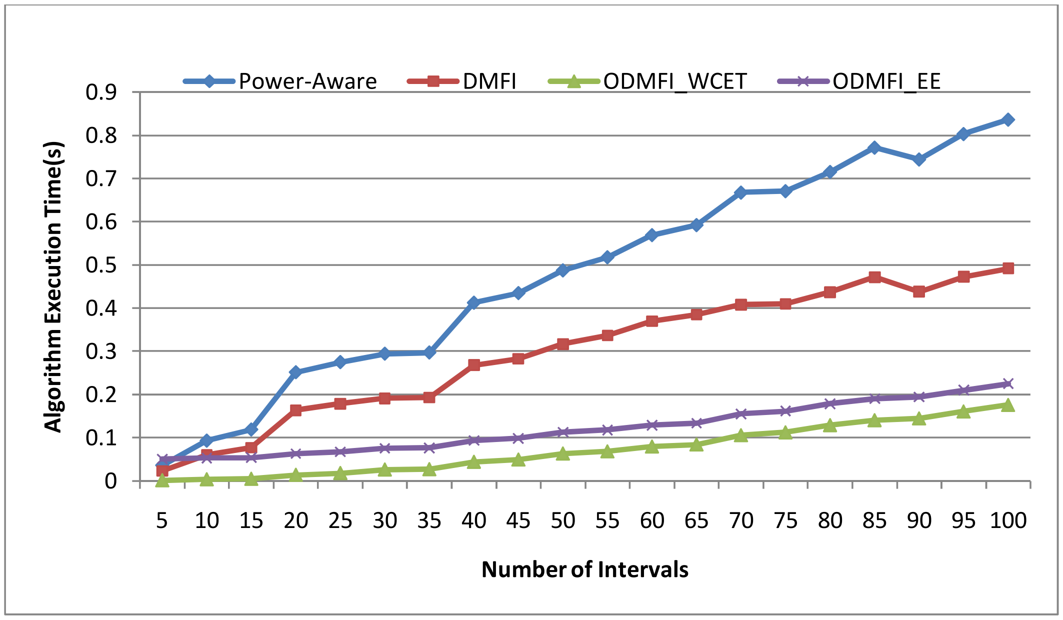

Figure 8. Algorithms comparison based on performance.

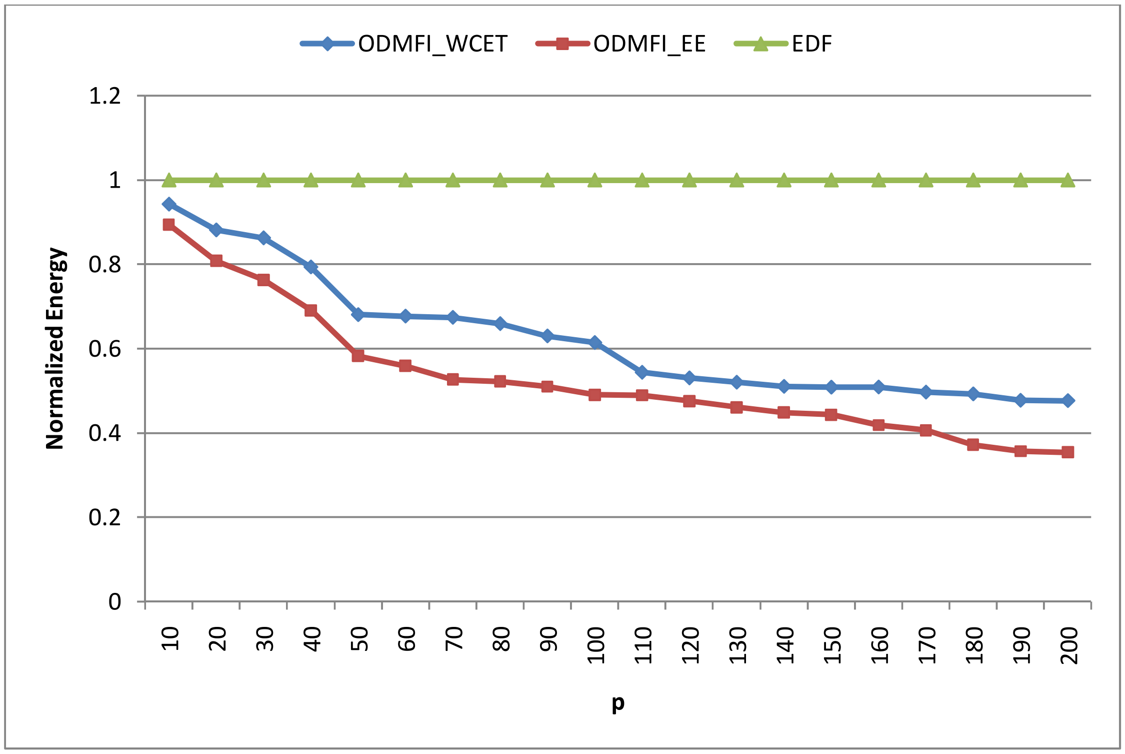

Figure 9. $\boldsymbol{p}$ value impacts on energy consumption.

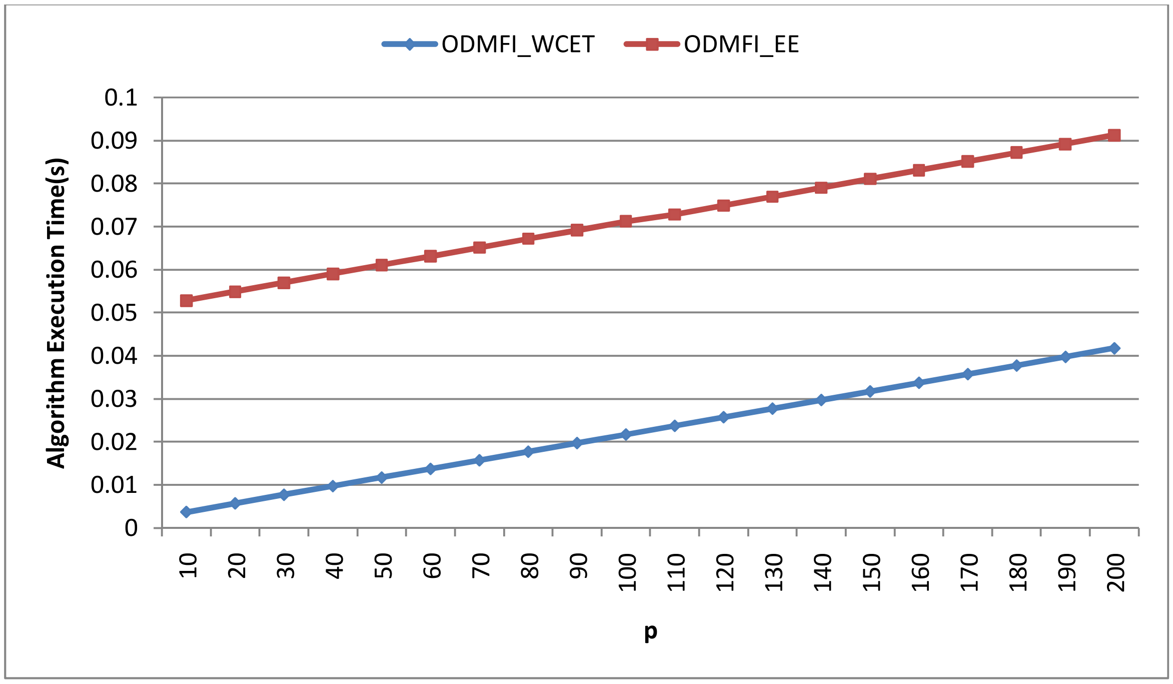

Figure 10. $\boldsymbol{p}$ value impacts on algorithm execution time.

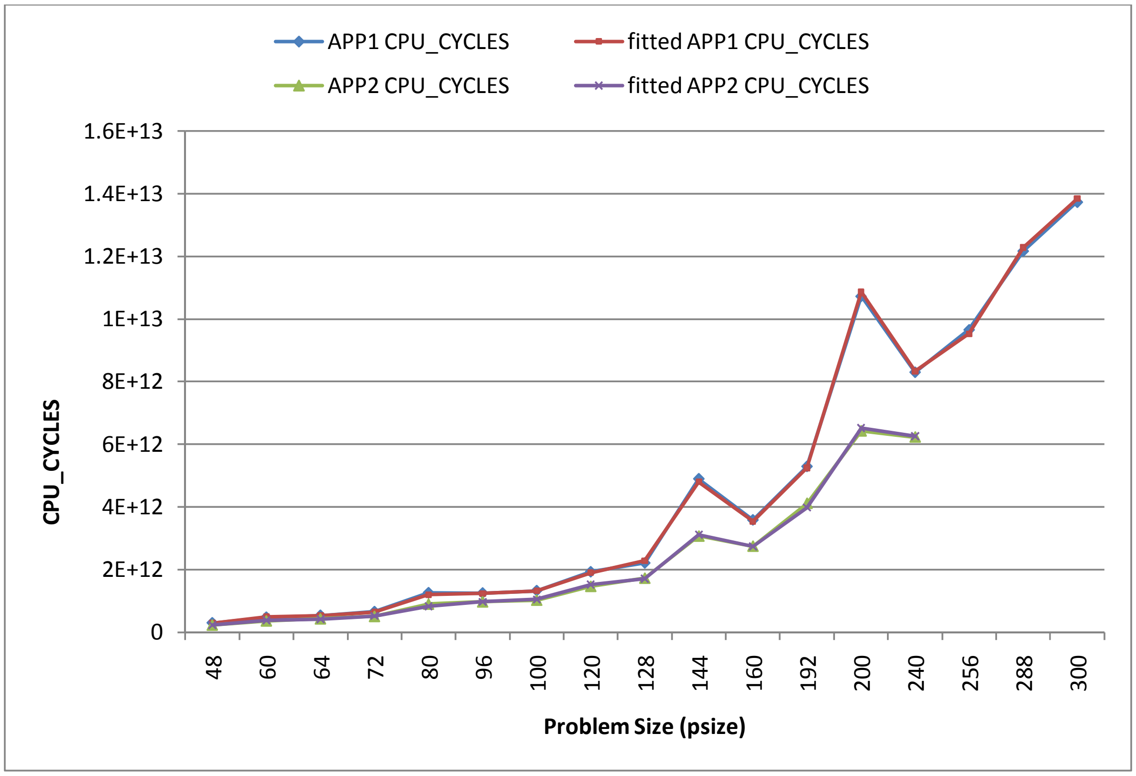

Figure 11. Prediction models from gls() curve fitting.

REFERENCES

[1] J.-J. Chen, J. Wu and C.-S. Shih, “Approximation algorithms for scheduling real-time jobs with multiple feasible intervals,” Journal of Real-Time Systems, pages 155-172, vol. 34, no. 3, Nov. 2006.

[2] C.-S. Shih, J. W.-S. Liu and I. K. Cheong, “Scheduling jobs with multiple feasible intervals,” *RTCSA*, 2004.

[3] Jian (Denny) Lin and Albert M. K. Cheng, “Power-aware scheduling for Multiple Feasible Interval Jobs,” *Proc. 15th IEEE-CS International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Beijing, China, Aug. 2009.

[4] Jian (Denny) Lin and Albert M. K. Cheng, “Maximizing Guaranteed QoS in (m,k)-firm Real-time Systems,” *Proc. 12th IEEE-CS International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, Sydney, Australia, pp. 402-410, Aug. 2006.

[5] Intel PXA255 Processor Data Sheet, www.phytec.com /pdf/datasheets/PXA255_DS.pdf .

[6] Jonathan Hall, Jian (Denny) Lin, and Albert M. K. Cheng, “Dynamic Multiple Feasible Intervals,” *Proc. IEEE-CS Real-Time and Embedded Technology and Applications Symposium (RTAS) WIP Session*, Stockholm, Sweden, April 13-16, 2010.

[7] Giorgio C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 2nd Edition, Springer 2005, 24-25, 92-94.

[8] Wei Song, *RealEnergy: a New Framework to Evaluate Power-Aware Real-Time Scheduling Algorithms*, Master thesis, University of Houston 2009.

[9] T. D. Burd and R. W. Brodersen, “Energy efficient CMOS microprocessor design”, In *Processing of the 2th Annual Hawaii International Conference on System Sciences*. Volumn 1: Architecture (Los Alamitos, CA, USA, Jan. 1995), T. N. Mudge and B. D. Shriver, Eds., IEEE Computer Society Press, pp. 288-297.

[10] C.-S. Shih, J.W.-S. Liu and I.K. Cheong, “Scheduling jobs with multiple feasible intervals”, In *RTCSA*, 2004.

[11] David Snowdon, Sergio Ruocco and Gernot Heiser, “Power Management and Dynamic Voltage Scaling: Myths and Facts”, In *Proceedings of the 2005 Workshop on Power Aware Real-time Computing*, New Jersey, USA, September, 2005.

[12] 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 International Real-Time Systems Symposium (RTSS’04)*, 2004, pp.319-327.

[13] TIBCO Software Inc., “TIBCO Spotfire S+ 8.1 Guide to Statistics, Volume I”, November 2008.

[14] Cachebench, http://icl.cs.utk.edu/projects/llcbench/cache bench.html.

[15] MiBench, http://www.eecs.umich.edu/mibench/.

[16] Gilberto Contreras, “Power prediction for Intel XScale processors using performance monitoring unit events”, In *Proceedings of the International symposium on Low power electronics and design (ISLPED05),* 2005, pp. 221-226.

[17] Intel XScale® Microarchitecture for the Intel® PXA255 Processor User’s Manual, order number 278796.