Maximizing Common Idle Time on Multi-core Processors with Shared Memory

Chenchen Fu*, Yingchao Zhao†, Minming Li*, Chun Jason Xue*
*Department of Computer Science, City University of Hong Kong, Hong Kong
†School of Computing and Information Sciences, Caritas Institute of Higher Education, Hong Kong

Abstract—**Reducing energy consumption is a critical problem in most of the computing systems today. This paper focuses on reducing the energy consumption of the shared main memory in multi-core processors by putting it into sleep state when all the cores are idle. Based on this idea, this work presents systematic analysis of different assignment and scheduling models and proposes a series of scheduling schemes to maximize the common idle time of all cores. An optimal scheduling scheme is proposed assuming the number of cores is unbounded. When the number of cores is bounded, an efficient heuristic algorithm is proposed. The experimental results show that the heuristic algorithm works efficiently and can save as much as 25.6% memory energy compared to a conventional multi-core scheduling scheme.**

I. Introduction

Energy efficiency is a critical issue in most of the computing environments nowadays. Among all the energy consuming components, main memory is one of the most significant energy consumers in mobile devices, servers, and computing systems. It is reported that main memory contributes to about 30-40% of total energy consumption on modern systems [6]. Leakage power occupies a significant portion of overall memory energy consumption, as the memory chips are becoming denser with smaller technology scales. Effectively reducing the leakage power becomes an important problem for memory energy efficiency. Nowadays, the main memory is usually shared by multiple cores in servers, personal computers, and even the embedded systems. Samsung Galaxy S4, which is equipped with eight-cores, is one example of multi-core processors applied in mobile devices [4]. Since cores have different memory access time according to the tasks executed in them, all the cores' activities should be taken into consideration when analyzing the memory energy consumption. In this work, we propose techniques to reduce the memory energy by orchestrating cores' activities, and turning the memory into sleep state as much as possible.

There are existing work focusing on the single-core processor based system. The memory can be turned into sleep state when it is not accessed by the only core [9][10]. The sleep mode transformation problem is more complicated with multiple cores with shared memory. Each core may have specific memory access pattern and the shared memory cannot sleep as long as the memory access exists. Consequently, it is the common idle time of all the cores in the system that determines the sleep time of the shared memory. In this paper, we explore how to maximize the common idle time to save the memory energy.

A similar problem related to maximizing common idle time was studied in a recent work [5]. In their model, the processor can handle up to B tasks in parallel, which is akin to the number of cores in our problem. Their objective is to minimize the total busy time of processors, which is equal to maximizing the common idle time. Focusing on the multi-interval task, which means that each task has a set of disjoint time periods to be scheduled, the authors formulate the problem by LP (Linear Programming) assuming arbitrary time preemption, prove the NP-hardness of the problem assuming only integral preemption allowed when the parallelism parameter $B \geqslant 3$, and propose efficient algorithms when $B = 2$. Even though the general case problem ($B \geqslant 3$) is proved to be NP-hard for multi-interval tasks, it remains interesting for the single interval tasks, which means that the task's feasible region, where the task is legal to be scheduled, is a continuous time period. In this paper, by considering single-interval tasks, optimal and efficient algorithms are proposed for the general case (the number of cores $\geqslant 2$).

In this paper, we conduct a systematic study of the common idle time maximization problem. The goal is to assign and schedule tasks among cores to maximize the time when all cores are idle, so that the memory sleep time can be maximized. Both theoretical and practical techniques are proposed in this paper. Experimental results show that the proposed heuristic algorithm performs close to the optimal solution. The main contributions of this paper are:

- The subproblems in the domain of common idle time maximization are categorized;
- When assuming the number of cores is unbounded, an optimal algorithm is proposed to maximize the common idle time;
- Considering a more general model that the number of cores is bounded by the number of tasks, an efficient heuristic algorithm is developed.

The rest of this paper is organized as follows. Section II presents the definitions of the system model and the target problem. In Section III, an optimal scheme and a heuristic algorithm are proposed. Experimental results are presented in Section IV. Finally we conclude the paper in Section V.

II. System Model And Problem Definition

In this section, we present the system and task models, together with the problem formulation.

System model: This paper explores energy-efficient scheduling schemes for the shared main memory over multi-core processors. Assuming the number of cores is C, each core can load only one task at a time. The main memory can be turned into sleep state when no core is accessing it. To clearly state the features of the target problem, in this work, we assume the sleep and active mode transition of the memory can be done instantly, and no extra energy overhead is required.

Task model: Tasks discussed in this paper are preemptive, single-interval and independent during executions. Given a task set $U = \{T_1, T_2, ..., T_n\}$, each task T_i is associated with release time r_i, deadline d_i, and non-negative processing time p_i. Note that single interval means that task T_i can only be feasibly scheduled during $[r_i, d_i]$, named as its feasible region.

Problem definition: To minimize the energy consumption of memory, this paper targets maximizing the memory sleep time, which is equivalent to the common idle time of cores. The *common idle time* is the time period when all cores are idle. The *busy interval* is defined as the maximal interval when at least one core is working. Based on the above model, we define the target problem as Maximizing Common Idle Time (MCIT) problem. In the following, MCIT problem is discussed over different conditions, as summarized in Table I.

In Table I, $C \geqslant n$ implies that the number of cores is sufficient for loading each task on a different core. When $C < n$, the schedulablity of tasks needs to be considered. *Length* is defined as the processing time of a task. *Arbitrary length* tasks allow preemption to happen at any point. However, since the processor works in cycles, for a more practical consideration, unit length time slot, which represents the length of scheduling quanta, needs to be considered and applied. Thus, *Unit length* and *Integral length* are defined for the tasks that can only begin, end and be preempted at endpoints of the unit slots. In this paper, different algorithms are proposed to solve the target problem both theoretically and practically.

III. Problem Analysis

In this section, we firstly discuss the problem when the number of cores is sufficient, and propose an optimal solution in Section III.A, and then develop a heuristic algorithm considering the number of cores is bounded in Section III.B.

A. Optimal solutions for MCIT with $C \geqslant n$

In this section, we present an optimal algorithm, Latest-Executing-Point-Driven Algorithm (LEPDA), which is inspired by the Lazy algorithm for unit length tasks in [5]. LEPDA can be applied for both arbitrary and integral length tasks.

LEPDA is an iterative algorithm. In each iteration, a task T_i with minimum $d_i - p_i$ among all tasks is found. $d_i - p_i$ represents the latest executing point of a task to meet its deadline. Since during the interval $[d_i - p_i, d_i)$, at least one core, on which task T_i is assigned, has to keep active, we set $[d_i - p_i, d_i)$ as the busy interval and assign other overlapping

TABLE I: Subproblems of MCIT considering the number of cores and task length. * *represents the work done in this paper.*

No. of Cores	Task Length	Solution
$C \geqslant n$	Unit length	Trivial
	Integral length	LEPDA*
	Arbitrary length	LEPDA*
$C < n$	Unit length	Lazy[5]/Dynamic Programming[7]
	Integral length	ILP*/LLFAA*
	Arbitrary length	LP[5]

Latest-Executing-Point-Driven Algorithm (LEPDA)

```
Input: $U = \{T_1, T_2, ..., T_n\}$;
1: while $U \neq \phi$ do
2:     Find $T_i$ with the minimum $d_i - p_i$ in $U$, and set $[d_i - p_i, d_i)$ as the
       busy interval;
3:     Let $D \subseteq U$ denote the set of tasks which have overlap with
       $[d_i - p_i, d_i)$;
4:     Assign tasks in $D$ within the busy interval as many as possible;
5:     Update $p_j$ for $\forall T_j \in D$ and remove the completed tasks from $U$;
6: end while
```

tasks in this interval as much as possible to maximize the benefit. Here, overlap means that the task's feasible region has overlap with the interval. If there are more than one task having the same latest executing point, we can choose the task with the latest deadline to minimize the number of iteration loops. For each task T_j that has been assigned to this interval, update its p_j or remove the task from U if completed. After each iteration, an interval $[d_{i-1}, d_i)$ is indicated, with $[0, d_1)$ being the first interval specifically. Each interval $[d_{i-1}, d_i)$ consists of two parts: a busy interval $[d_i - p_i, d_i)$ and an idle interval $[d_{i-1}, d_i - p_i)$. We call T_i, which is found in each iteration, the "critical task". The total length of all the idle intervals is the common idle time we try to maximize.

We can see that the schedule generated by LEPDA is feasible. For the optimality proof of LEPDA, Lemma 1 is proposed to show that there exists an optimal solution satisfying Lemma 1, and then Theorem 1 can show that the optimal solution given in Lemma 1 needs at least the same busy time as LEPDA does.

Lemma 1. *There exists an optimal solution in which each busy interval ends with the deadline of a critical task.*

Theorem 1. *The optimal solution for MCIT problem with $C \geqslant n$ can be found by LEPDA.*

Proofs are omitted here due to the space limitation.

B. Heuristic algorithm for MCIT with $C < n$

In this subsection, the schedulablity of tasks is taken into account as the number of cores is bounded by the number of tasks. Chang et al. propose LP to solve this problem when assuming preemption is arbitrarily allowed [5]. Considering the problem for integral length tasks (as defined in Table I) is more practical and important in reality, in the following, we propose a heuristic scheme, Least-Laxity-First Assigning Algorithm (LLFAA) for integral length tasks. In this subsection, we define slot (or unit slot) as the minimal non-preemptive scheduling time unit.

LLFAA is not optimal, as tasks need to be guaranteed to be feasibly scheduled before maximizing the common idle time. However, currently no effective polynomial algorithm has been developed to obtain the optimal schedulablity of tasks among multi-cores. In LLFAA, we apply the Least Laxity First (LLF) scheme, which is a better scheme [8] for schedulablity of tasks among multi-core system than EDF (Earliest Deadline First), to maximize the schedulablity of tasks. LLF schedules tasks in the order of increasing laxity, which refers to $(d_i - t^c) - p_i^r$, where p_i^r represents the remaining processing time of T_i and t_c represents the current time.

LLFAA begins by finding a busy interval in each iteration. Then a value x_j, which means the least processing time of $T_j \in D$ that should be executed in the interval $[d_i - p_i, d_i)$ to meet the deadline needs to be calculated. Procedure LLFUSA

Least-Laxity-First Assigning Algorithm (LLFAA)

```
Input: $U_0 = U = \{T_1, T_2, ..., T_n\}$;
1: while $U \neq \phi$ do
2:     Find $T_i$ with minimum $d_i - p_i$ in $U$, and set $[d_i - p_i, d_i)$ as the busy
       interval;
3:     Let $D \subseteq U_0$ deote the set of tasks which have overlap with
       $[d_i - p_i, d_i)$;
4:     Let $x_j = \max(0, p_j - (d_j - d_i))$, and execute Procedure LLFUSA;
5:     if The returned result of LLFUSA is not $\phi$ then
6:         Do binary search to find the appropriate $m$ extra busy slots in the
           left of the current busy interval;
7:     end if
8:     Update $p_j$ for $\forall T_j \in D$ and remove the completed tasks from $U$;
9:     if The final result of LLFUSA is $\phi$ then
10:        Let $x_j = p_j$, and execute Procedure LLFUSA;
11:        Update $p_j$ for $\forall T_j \in D$ and remove the completed tasks from $U$;
12:    else
13:        return Fail to schedule;
14:    end if
15: end while
```

Procedure: LLF-based Unit Slot Assignment (LLFUSA)

```
1: For each slot in the busy interval $[x, y)$, if the slot is idle on some core,
   assign tasks to idle cores in the increasing order of $\min(d_j, y) - x_j$, and
   update $x_j = x_j - 1$ if $T_j$ has been assigned;
2: return Tasks with $x_j > 0$;
```

is an algorithm that assigns each given task's x_j to the current busy interval using LLF scheme. After executing LLFUSA, LLFAA checks whether all tasks' x_j slots have been assigned. If yes, all tasks can be successfully scheduled. Otherwise, the length of the tentative busy interval needs to be extended to obtain a feasible schedule. Binary search is applied to find an appropriate length of idle slots to add to the tentative busy interval. These idle slots are chosen from all the idle slots on the left side of the tentative busy interval. Let the chosen idle slots be busy. Note that if the tentative busy interval is attached with the former busy intervals, we treat these slots as a new busy interval and reassign tasks to it. After the binary search, it will either obtain a feasible busy interval, or a terminal message of failure to schedule. As some slots in this busy interval may still be idle, LLFUSA is processed again to assign tasks' remaining processing length to those idle slots, to maximize the utilization of the busy interval.

Let $P = \sum p_i$ denote the sum of processing time of tasks, the time complexity of LLFAA is $\mathcal{O}(nPlog(P)Clog(C))$. Note that $P = \sum p_i$ indicates that LLFAA is pseudo-polynomial. However, experimental results show that it is an efficient algorithm in practice, performing close to the optimal solution.

IV. Evaluations

In this section, we evaluate the proposed heuristic algorithm LLFAA. To illustrate the effectiveness, we develop an optimal solution, obtained by ILP (Integer Linear Programming), and then compare LLFAA with the optimal solution. The ILP formulation is omitted here due to the space limitation. The proposed scheme is also compared with the algorithm LLF [8] to show the potential improvement of energy saving that can be achieved. Experiments are conducted with both real benchmarks and synthetic applications.

Real benchmarks are obtained from Embedded System Synthesis Benchmark Suite (E3S) [1] and Mediabench [2]. Each application consists of a set of tasks, which are scheduled in a six-core processor. The deadline of a task T_i is set to be a random value in $[r_i + p_i, r_i + 3p_i]$. For each benchmark, we obtain its task dependencies and construct the data flow graph. The dependency among tasks are guaranteed by setting the release time of a task as the maximum deadline of all the incoming tasks.

The scheduling results of idle time over schedule length ratio for different algorithms are given in Fig. 1. LLFAA performs very close to the optimal solution, and is much better than LLF, for most of the benchmarks. Note that all algorithms have the same idle time for *mesa-osdemo* and *gsm-untoast*. This is because there are long running tasks dominating the whole schedule, and thus different scheduling schemes have similar common idle time.

For real benchmarks, the number of tasks ranges from 10 to 29, which are quite small task sets. In order to evaluate the proposed algorithm with large task sets, evaluations with synthetic applications are conducted as shown in the following. ILP solution is computationally intensive, which cannot get the results within reasonable time when the size of time slots or the number of tasks is large. In order to draw the conclusions clearly, the experiments are evaluated based on 2 different sizes of slots number s: regular size $s = 100$ and large size $s = 1000$. In the setting of the synthetic tasks, we assume that tasks have 60% probability to be randomly released in $[0, s/2]$ and 40% probability in $[s/2 + 1, s]$, because release time is more likely to appear in the first half of the time interval to guarantee that tasks are uniformly distributed. Task's deadline, d_j, is randomly generated between $[r_j, s]$. If the length of a task's feasible region $d_j - r_j$ is less than $s/2$, it will be regarded as a regular length task. Otherwise, it is regarded as a long length task. Each task is set to have 60% probability to be a regular length task and 40% probability to be a long length task.

In the experiments of synthetic applications, all the algorithms are evaluated across three parameters: *(i)* The density of tasks, which is the number of tasks in the fixed time interval $[0, s]$; *(ii)* The demand of tasks, which is the ratio $\frac{p_j}{d_j - r_j}$; *(iii)* The number of cores. For each parameter, we list nine data points for regular and large cases respectively, as shown in TABLE II. For each data point, we randomly generate ten different cases, and use the average value as the final performance of each data point. For each size, the value marked with a * represents the default value when algorithms are evaluated over the other parameters. The detailed performance evaluation results over various parameters are shown in Fig. 2 and Fig. 3. The vertical axis stands for the sleep time ratio for all figures.

The results for the regular case are shown in Fig. 2. In Fig. 2(a), when the tasks' density is small, LLFAA performs very close to the optimal solution. The performance improvement

TABLE II: Different parameters for evaluating algorithms.

Point	Density of Tasks		Demand of Tasks	Number of Cores	
	Regular	Large		Regular	Large
1	20	200	<0.15	4	40
2	30	300	<0.2	6	50
3	40	400	<0.25	8	60
4	50	500	<0.3	10	70
5	60*	600*	<0.35	12*	80*
6	70	700	<0.4	14	90
7	80	800	<0.45	16	100
8	90	900	<0.5*	18	110
9	100	1000	>0.5	20	120

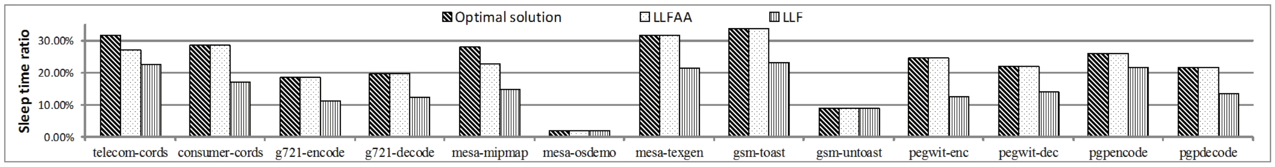
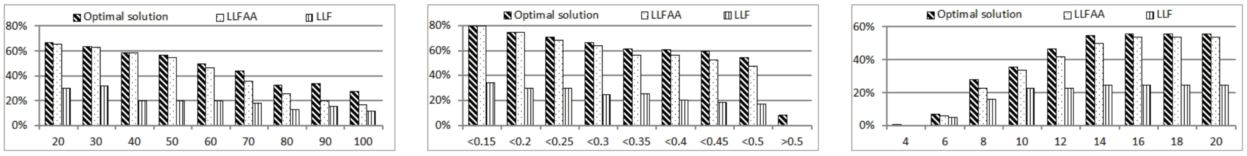
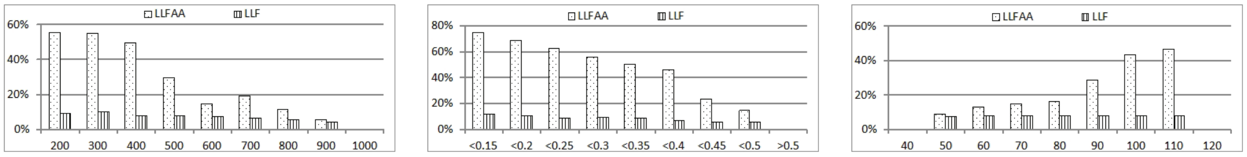


Fig. 1: Comparison of the common idle time (memory sleep time) over schedule length for Real Benchmarks.



(a) Density of tasks. (b) Demand of tasks. (c) Number of cores.

Fig. 2: Evaluations of LLFAA, the optimal solution and LLF over 3 different parameters (s=100).



(a) Density of tasks. (b) Demand of tasks. (c) Number of cores.

Fig. 3: Evaluations of LLFAA and LLF over 3 different parameters (s=1000).

decreases with the increasing of the the density of tasks. In Fig. 2(b), LLFAA is very close to the optimal solution when tasks' processing demands are not so high. When the demand of tasks is larger than 0.5, in the ten cases that are randomly generated, only one of them could be feasibly scheduled by the optimal solution, while LLFAA and LLF fail to schedule all the cases. This shows that the proposed heuristic algorithm may fail to schedule a feasible case when all tasks' processing demands are high. For most cases, the proposed algorithm is efficient in practice.

The evaluation over different numbers of cores is shown in Fig. 2(c). Different from the above two evaluations, this time we keep the generated task sets of ten cases the same over the nine data points, while only changing the number of cores. In Fig. 2(c), the difference between LLFAA and the optimal solution decreases with the increasing of the number of cores. When the number of cores is 4, most of the cases are infeasible to be scheduled by the optimal solution. Only two cases are feasible. LLFAA fails to schedule for one case, and obtains zero idle slot for the other. It can be noted that LLFAA performs better when the number of cores is sufficient.

The other set of tasks with large size slots are shown in Fig. 3. As the time for obtaining the optimal solution is not acceptable for the large case, we only compare LLFAA with LLF to evaluate the benefits obtained by LLFAA. As shown in Fig. 3(a) and Fig. 3(c), the differences between the proposed solution and LLF are more significant than the regular case.

LLFAA is an efficient and effective algorithm in maximizing the memory sleep time. For the energy consumption analysis, we assume that memory working in active mode consumes two times more energy than working in sleep mode [3]. For real benchmarks, the average energy saving improvement is 5.3% (up to 8.6%) compared to LLF. The improvement is not quite significant because there are few differences between LLF and the optimal solution (the optimal solution can only obtain 5.8% improvement). However, when the workload is not heavy, as shown by the large synthetic tasks set, LLFAA can reduce the memory energy by 25.6% in average.

V. Conclusion

In order to reduce the energy consumption of the memory in a multi-core architecture, this paper proposes scheduling schemes to maximize the memory sleep time, which is equivalent to the common idle time of all cores. When the number of cores is not bounded, an optimal solution is proposed; when the number of cores is bounded, an efficient heuristic algorithm is developed. Evaluations show that the proposed heuristic algorithm performs close to the optimal solution and can effectively reduce the memory energy consumption.

Acknowledgement

This work was partially supported by grants from City University of Hong Kong [Project No. CityU 117913] and [Project No. CityU 9231168].

References

[1] E3s. http://ziyang.eecs.umich.edu/∼dickrp/e3s/.
[2] Mediabench. http://euler.slu.edu/∼fritts/mediabench/.
[3] Micron technology inc., "tn-47-04: Calculating memory system power for ddr2". http://www.micron.com/support/dram/power_calc.html.
[4] Samsung galaxy s4. http://www.samsung.com/global/microsite/galaxys4/.
[5] J. Chang, H. N. Gabow, and S. Khuller. A model for minimizing active processor time. ESA, pages 289–300, 2012.
[6] G. Dhiman, R. Ayoub, and T. Rosing. Pdram: A hybrid pram and dram main memory system. DAC, pages 664–469, 2009.
[7] G. Even, R. Levi, D. Rawitz, B. Schieber, S. M. Shahar, and M. Sviridenko. Algorithms for capacitated rectangle stabbing and lot sizing with joint set-up costs. *ACM Trans. Algorithms*, 4(3):34:1–34:17, July 2008.
[8] R. G. Herrtwich. An introduction to real-time scheduling. Technical report (International Computer Science Institute), TR-90-035, July 1990.
[9] C.-G. Lyuh and T. Kim. Memory access scheduling and binding considering energy minimization in multi-bank memory systems. DAC, pages 81–86, 2004.
[10] Z. Wang and X. S. Hu. Energy-aware variable partitioning and instruction scheduling for multibank memory architectures. *ACM Trans. Des. Autom. Electron. Syst.*, 10(2):369–388, April 2005.