2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops

EEWA: Energy-Efficient Workload-Aware Task Scheduling in Multi-core Architectures

Quan Chen*, Long Zheng*†, Minyi Guo*, Zhiyi Huang‡
*Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China
†University of Aizu, Japan
‡Department of Computer Science, University of Otago, New Zealand
chen-quan@sjtu.edu.cn, longzheng@sjtu.edu.cn, guo-my@cs.sjtu.edu.cn, zhuang@cs.otago.ac.nz

***Abstract*—Modern multi-core architectures offer Dynamic Voltage and Frequency Scaling (DVFS) that can dynamically adjust the operating frequency of each core for energy saving. However, current parallel programming environments and schedulers for task-based programs do not utilize DVFS and thus suffer from energy inefficiency in multi-core processors. To reduce energy consumption while keeping high performance, this paper proposes an Energy-Efficient Workload-Aware (EEWA) task scheduler that is comprised of a workload-aware frequency adjuster and a preference-based task-stealing scheduler. Using DVFS, the workload-aware frequency adjuster can properly tune the frequencies of the cores according to the workload information of the tasks collected with online profiling. The preference-based task-stealing scheduler can then effectively balance the workloads among cores by stealing tasks according to a preference list. Experimental results show that EEWA can reduce energy consumption of task-based programs up to 29.8% with a slight performance degradation compared with existing task schedulers.**

***Keywords*-Online Profiling; DVFS; Task Scheduling**

I. Introduction

Many studies [17], [15] have shown that Asymmetric Multi-Core (AMC) architectures, where cores have different individual frequencies, are more energy-efficient. In AMC architectures, fast and complex cores (aka. big cores) can be used to execute the serial code sections, while slow and simple cores (aka. little cores) can be used to crunch numbers in parallel. If an application can use the big/little cores properly, energy consumption is found to be reduced.

However, it is challenging for an AMC architecture to achieve the best performance for all applications while consuming the lowest energy. To achieve optimal energy efficiency, different applications may need different number of big/little cores according their computational and memory-access patterns, an requirement that an asymmetric multi-core architecture with fixed big/little cores cannot meet.

Fortunately, many modern multi-core chips offer *Dynamic Voltage and Frequency Scaling* (DVFS) [24] which can dynamically adjust the operating frequency of each core at runtime. With DVFS, any core can be adjusted dynamically to run at the frequency required by an application.

Despite the effort on energy saving at the hardware level, current parallel programming environments and task scheduling algorithms [5], [9] cannot utilize DVFS effectively since they assume all cores run at fixed frequencies or the same frequencies during the execution of applications.

Most parallel programming environments adopt either task-sharing or task-stealing (aka. work-stealing) policies for task scheduling. For example, Cilk++ [19], TBB [27], and X10 [18] adopt task-stealing, while OpenMP [2] uses task-sharing. Task-stealing is increasingly popular due to its good scalability and high performance [4].

Both task-stealing and task-sharing work well in terms of performance on multi-core processors with cores operating at fixed frequencies. If the performance of an application is improved, the energy consumption is reduced due to the shorter execution time. Therefore, the current parallel programming environments are reasonably energy-efficient.

However, with DVFS support, we can further increase energy-efficiency for parallel applications. For example, suppose a parallel application only has parallel tasks γ_1 and γ_2 which are scheduled to cores c_1 and c_2 respectively. Assuming γ_1 and γ_2 need time t_1 and t_2 ($t_1 < t_2$) to run on the fastest core with the highest frequency, the application can finish in t_2 if both c_1 and c_2 are running at the highest frequency. However, if we can scale down c_1's frequency so that c_1 finishes executing γ_1 in t_2 as well, then the energy consumption of the application is reduced while its performance is not sacrificed at all. These situations are not taken into account by the current task scheduling algorithms.

Based on this observation, this paper proposes an *Energy-Efficient Workload-Aware* (EEWA) task scheduling to reduce energy consumption without degrading performance conspicuously. EEWA consists of two parts: a *workload-aware frequency adjuster* and a *preference-based task-stealing scheduler*. The workload-aware frequency adjuster tunes the frequencies of all the cores according to the workload information of tasks that is collected through online profiling for iteration-based parallel applications.

*Minyi Guo is the correspondence author of this paper.

978-1-4799-4116-2/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPSW.2014.75

The preference-based task-stealing scheduler is then adopted to balance workloads dynamically. Since the frequency adjuster can reduce energy consumption by scaling down the frequency of cores and the preference-based task-stealing scheduler can schedule tasks to the proper cores, EEWA can significantly reduce the energy consumption while only slightly degrading the overall performance.

The contributions of this paper are as follows.

- We propose a workload-aware frequency adjuster using online profiling information to dynamically search the proper configuration of the cores' frequencies that can reduce the energy consumption of the parallel application with little impact on the performance.
- We adopt a preference-based task-stealing scheduler to balance workloads among cores dynamically so that the performance of the parallel application will not degrade seriously even if the tasks are mis-allocated occasionally by the frequency adjuster.

The rest of this paper is organized as follows. Section II describes the problem and the solution of EEWA. Section III presents EEWA. Section IV evaluates EEWA and gives the experimental results. Section V discusses the related work. Section VI draws conclusions.

II. Problem and Solution

We use a small example to illustrate the situations where energy can be saved without affecting performance. Fig. 1 shows four possible schedules of tasks γ_0 and γ_1 of a parallel application on a dual-core processor with DVFS support. Suppose the cores may run at frequencies f_0 and $0.5f_0$ with power p_0 and p_1 ($p_0 > p_1$) respectively. Suppose γ_0 and γ_1 take times $2t$ and t on the core with frequency f_0 respectively. Assuming the tasks are CPU-bound, it is reasonable to deduce that γ_0 and γ_1 would take $4t$ and $2t$ on the core with frequency $0.5f_0$.

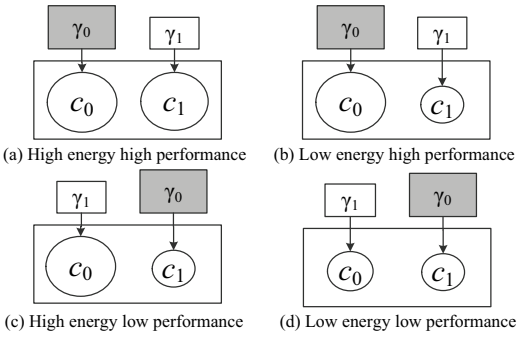


Figure 1. Four possible schedules of γ_0 and γ_1 on a dual-core processor with DVFS support.

In Fig. 1(a), both cores run at the highest frequency f_0. In Fig. 1(b) and Fig. 1(c), the frequency of c_1 is scaled down to $0.5f_0$ using DVFS. In Fig. 1(d), the frequencies of both cores are scaled down to $0.5f_0$. Then, the execution time and the energy consumed by the application in the four schedules are calculated as follows.

Fig. 1(a) shows the traditional way of task scheduling. In traditional task scheduling, such as task-sharing and task-stealing, the frequency of c_1 will not be scaled down after c_1 finishes γ_1 since c_1 will be actively trying to get or steal new tasks until the application terminates. Therefore, the overall execution time of the application is $\max\{2t, t\}$ = 2t and the energy consumption is $2 \times p_0 \times 2t = 4tp_0$.

In Fig. 1(b), the execution time is $\max\{2t, 2t\}$ = 2t and the energy consumption is $p_0 \times 2t + p_1 \times 2t = 2t(p_0 + p_1)$. Since $(p_0 + p_1) < 2p_0$, compared with the traditional scheduling in Fig. 1(a), the schedule in Fig. 1(b) reduces the energy consumption without degrading the performance. This is an optimal situation where EEWA aims to take advantage of.

In Fig. 1(c), the execution time is $\max\{4t, t\}$ = 4t and the energy consumption is $p_0 \times 4t + p_1 \times 4t = 4t(p_0 + p_1)$. This schedule seriously degrades the overall performance and increases the power consumption. Compared with Fig. 1(b), this is an unfortunate situation task schedulers should avoid. In EEWA, this situation is avoided by the preference-based task-stealing scheduler.

In Fig. 1(d), the execution time is $\max\{4t, 2t\}$ = 4t and the energy consumption is $2 \times p_1 \times 4t = 8tp_1$. This schedule also seriously degrades the overall performance and increases the power consumption.

In summary, Fig. 1(b) is the most energy efficient schedule. For the traditional scheduling algorithms such as task-stealing, energy is wasted as in Fig. 1(a) where workloads of the tasks are not the same. For example, in the task-stealing algorithm, the idle cores have to be busily trying to steal new tasks until all cores finish their tasks. If we can scale down the frequencies of the cores executing small tasks so that all tasks complete at the same time, as in Fig. 1(b), energy will be surely saved.

To achieve the optimal schedule in Fig. 1(b), two challenging issues have to be addressed. First, to reduce energy, a proper configuration of the cores' frequencies should be explored. For example, if an improper configuration is adopted as in Fig. 1(d), the performance of the application will be seriously degraded. To find out a proper configuration, information of task workloads have to be known. Second, to maintain the high performance, an efficient task scheduler is required to balance the workloads among cores. Fig. 1(c) is an example of inappropriate scheduling. Although the frequencies of the cores are adjusted the same as in Fig. 1(b), the performance is seriously degraded and the energy is wasted in Fig. 1(c) due to the poor scheduling scheme.

Based on the example in Fig. 1, the problem can be summarized as a task scheduling problem: *without complex offline analysis, how to schedule a set of parallel tasks with different workloads in an energy efficient manner at runtime on a multi-core architecture with DVFS support?*

A. The proposed solution

We propose the *Energy-Efficient Workload-Aware* (EEWA) task scheduling to solve the problem for iteration-based applications where parallel tasks are executed in batches through iterations. EEWA is based on the assumption that the task workloads of different iterations have similar patterns. Based on this assumption, if a configuration of the cores' frequencies is optimal for one iteration, it is very likely to be optimal for other iterations. Therefore, when tasks in one iteration complete, with the online profiling information collected, EEWA can calculate the proper configuration of the cores' frequencies for the execution of the following iteration.

EEWA adopts a workload-aware frequency adjuster to find out the proper configuration of the cores' frequencies. Take a parallel application running on a multi-core processor with m cores as an example. Suppose each core has r different frequencies. When an iteration I_d of the application completes, the adjuster collects the number of tasks and their workloads during I_d. The tasks are then grouped into *task classes* according to their function names. With the workload information of the task classes, an r-row k-column *Core Count* (CC) table can be built, where k is the number of task classes. The element CC_{ji}, represents the required number of cores for the task class i to be executed by the cores with the frequency j. With the CC matrix, the frequency adjuster can find out an optimal configuration of the cores' frequencies for the iteration I_d using a backtracking algorithm. The configuration will be used for setting the cores' frequencies for the following iteration. Based on the configuration, the cores are organized into c-groups according to their frequencies and the task classes from the following iteration are allocated to the proper c-groups according to the CC matrix. Note that a *c-group* is a set of cores with the same operating frequency.

However, we should admit that the workloads of tasks may change slightly in different iterations. In such a case, the allocation by the frequency adjuster may not balance workloads among cores very well. To further balance workloads, EEWA adopts the preference-based task-stealing scheduler to balance workloads dynamically. The scheduler gives each core a preference list of c-groups. An idle core steals a task according to the order of its preference list.

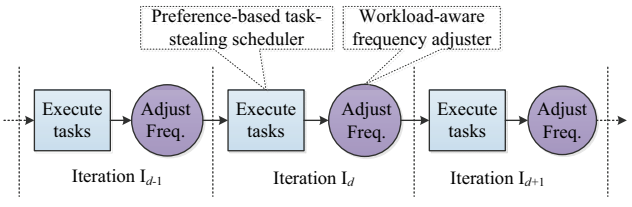


Figure 2. The processing flow of a parallel program in EEWA.

Fig. 2 illustrates the processing flow of an iteration-based parallel application in EEWA. As shown in the figure, once EEWA completes tasks in iteration I_d, the workload-aware frequency adjuster is launched. Based on the collected workload information of tasks in iteration I_d, EEWA can find out the proper configuration of the cores' frequencies and adjust the frequencies of all the cores for iteration I_{d+1}.

III. Energy Efficient Workload-Aware Task Scheduling

We present the workload-aware frequency adjuster and the preference-based task-stealing scheduler in EEWA as follows. In this section, we suppose that EEWA operates on a multi-core processor with m cores with r operating frequencies, in which F_0, ..., F_{r-1} represent the r frequencies in descending order (i.e., $F_i > F_j$ if $i < j$).

A. Workload-aware frequency adjuster

For an iterative parallel application, at the end of each iteration, EEWA uses the workload-aware frequency adjuster to find out a proper configuration of the cores' frequencies for the next iteration in the three steps: collect workload information, build CC table and search for the solution.

1) Collect workload information: In this step, the adjuster collects the number of tasks and their workloads. The workload of a task is measured with its execution time and normalized against the fastest core with frequency F_0. For a CPU-bound task, its execution time is largely decided by the computational capacity (i.e., the frequency) of the core executing the task. Suppose a CPU-bound task γ is executed by a core of frequency F_i and the execution time is t. Its normalized workload (denoted by w_γ) is calculated with Eq. 1.

$$w_\gamma = t \times \frac{F_i}{F_0} \tag{1}$$

As mentioned before, the completed tasks in the iteration I_d are organized as *task classes* according to their function names. We use $TC(f, n, w)$ to represent a task class, where f is the function name, n is the number of tasks in the task class and w is the average workload of the tasks.

Once a task γ with a function name f completes, its task class $TC(f, n, w)$ is updated. Suppose γ's workload is w_γ. Its task class is updated to $TC(f, n+1, \frac{n \times w + w_\gamma}{n+1})$. If there is no task class for function f yet, a new task class $TC(f, 1, w_\gamma)$ is created. When all tasks in I_d complete, they are grouped into a number of task classes.

EEWA does not adjust the frequencies of the cores during the first iteration of the parallel application, since the workload information of tasks are not known yet. Therefore, in the first iteration, all the cores run at the highest frequency F_0. The execution time of the first iteration is set as the *ideal iteration time* which is targeted by the following iterations, assuming the overall workload of each iteration is similar.

Let T represent the execution time of the first iteration. To keep the same performance, EEWA aims to finish each of the following iterations in T.

2) Build CC Table: For the iteration I_d, suppose k task classes $TC_0(f_0, n_0, w_0)$, ..., $TC_{k-1}(f_{k-1}, n_{k-1}, w_{k-1})$ are created, where w_i $(0 \leq i \leq k-1)$ is in descending order. Based on the k task classes and the ideal iteration time T, we can build a CC table in Table I. We use CC_{ji} to represent one element of the table at row F_j and column TC_i. CC_{ji} is the number of cores operating at frequency F_j that is needed to finish all the tasks in TC_i within T. The adjuster calculates CC_{ji} $(0 \leq j \leq r-1,\ 0 \leq i \leq k-1)$ as follows.

For TC_i (f_i, n_i, w_i), the overall workload of the tasks in the task class is $n_i \times w_i$. Therefore, $\frac{n_i \times w_i}{T}$ is the number of cores required to finish executing the tasks within T, if the cores are operating at the highest frequency F_0. In another word, CC_{0i} should be $\frac{n_i \times w_i}{T}$. Naturally, since the computation capacity of a core with F_0 is $\frac{F_0}{F_j}$ times of the computation capacity of a core with F_j, CC_{ji} should be set to $\frac{F_0}{F_j} \times \text{CC}_{0i} = \frac{F_0}{F_j} \frac{n_i \times w_i}{T}$.

Table I
THE CC TABLE FOR k TASK CLASSES ON THE MULTI-CORE PROCESSOR WITH r OPERATING FREQUENCIES

	TC_0	TC_1	...	TC_{k-1}
F_0	$\frac{n_0 \times w_0}{T}$	$\frac{n_1 \times w_1}{T}$	...	$\frac{n_{k-1} \times w_{k-1}}{T}$
F_1	$\frac{F_0}{F_1} \frac{n_0 \times w_0}{T}$	$\frac{F_0}{F_1} \frac{n_1 \times w_1}{T}$	...	$\frac{F_0}{F_1} \frac{n_{k-1} \times w_{k-1}}{T}$
...	...	...	...	...
F_{r-1}	$\frac{F_0}{F_{r-1}} \frac{n_0 \times w_0}{T}$	$\frac{F_0}{F_{r-1}} \frac{n_1 \times w_1}{T}$	...	$\frac{F_0}{F_{r-1}} \frac{n_{k-1} \times w_{k-1}}{T}$

3) Search for the solution: Finding out a proper configuration of the cores' frequencies from the CC table is similar to searching for a solution in a number puzzle. In this number puzzle, a *k-tuple*, which consists of k numbers, should be found based on the CC table.

The k-tuple, denoted by $(a_0, ..., a_i, ..., a_{k-1})$, corresponds to the k columns of the CC table from left to right, where a_i $(0 \leq i \leq k-1)$ means the element $\text{CC}_{a_i i}$ is selected. One and only one element in each column of the CC table is selected for the k-tuple. If CC_{ji} is selected, tasks in TC_i are supposed to be executed on cores with frequency F_j.

However, it is challenging to find for the number puzzle an optimal solution that can minimize the energy consumption while maintaining the high performance. If too many cores are adjusted to run at high frequencies, the energy consumption cannot be reduced. On the other hand, if too many cores are adjusted to run at low frequencies, the performance would be seriously degraded.

To find an optimal solution, the k-tuple and the search algorithm should satisfy the following three constraints. First, the sum of the selected elements should be less or equal to the total number of cores, i.e., $\sum_{i=0}^{k-1} CC_{a_i i} \leq m$ where m is the total number of cores. Otherwise, the frequencies are adjusted too low and the performance will be degraded. Second, the search should start from the bottom of the CC table. Since we are trying to minimize the energy consumption, we should search for the solution with the lowest frequencies that can satisfy the first constraint. Third, the k-tuple should meet the condition $a_i \leq a_j$ if $i < j$. Since tasks in TC_i have heavier workload than those in TC_j when $i < j$, execute heavier tasks on faster cores can better utilize the computational capacity of the faster cores.

Algorithm 1: Backtracking algorithm for the k-tuple

```
Input: CC table CC[r][k] (k task class, r freq.)
Input: N (the total number of cores)
Output: a[k] (the k-tuple)
1  int c_n = 0 ;      // Required num. of cores
   Func.: Select(i, j)
2  if CC[j][i] + c_n ≤N then
3  |  a[i] = j; c_n += CC[j][i] ;
4  |  return true ;
5  end if
6  return false ;
   Func.: SearchTuple(i)
7  if i>=k then
8  |  return true ;
9  else
10 |  for (int j=r-1;j>=a[i-1];j--) do
11 |  |  if Select(i,j) then
12 |  |  |  if SearchTuple(i+1) then
13 |  |  |  |  return true ;
14 |  |  |  else
15 |  |  |  |  c_n −=CC[a[i]][i]
16 |  |  |  end if
17 |  |  end if
18 |  end for
19 |  return false ;
20 end if
```

Algorithm 1 is a backtracking algorithm that can find the k-tuple satisfying the three constraints. The workload-aware frequency adjuster invokes *SearchTuple(0)* to search for an optimal k-tuple. The reason we choose this backtracking algorithm is that it can give a near-optimal solution with reasonable overhead. The worst-case complexity of this algorithm is $O(kr^2)$. Although other algorithms (such as exhaustive search) may be used to find more optimal solutions, they are either too complex to implement or have large overhead.

Fig. 3 shows an example of the k-tuple. In the example, there are 4 task classes (TC_0, ..., TC_3) and each core can run at 4 different frequencies(F_0, F_1, F_2 and F_3). Suppose there are 16 cores in total. Based on the CC table in Fig. 3, Algorithm 1 selects the shaded elements that correspond to the k-tuple (1, 1, 2, 2). According to the k-tuple and the CC table in Fig. 3, 10 cores should run at frequency F_1,

and 6 cores should run at frequency F_2. Moreover, TC_0 and TC_1 should run on the fast cores, and TC_2 and TC_3 should run on the slow cores. Other solutions either cannot execute all the tasks in the ideal iteration time with the 16 cores or consume more energy due to the faster cores selected.

	TC_0	TC_1	TC_2	TC_3
F_0	2	3	1	1
F_1	4	6	2	2
F_2	6	9	3	3
F_3	8	12	4	4

Search from lower frequencies

Search from task class with heavier workloads

Figure 3. An example of the k-tuple for a parallel program with 4 task classes on a 16-core processor with 4 frequencies.

Careful readers may observe that, from Fig. 3, the sum of core counts (7) in the first row is much smaller than the total number of cores (16). The reason is the workload imbalance causes the underutilization of the computational capacity of the cores, though the gap should not be so large in practice. This is why, by scaling down the frequencies of some cores, EEWA can better balance the workloads among the cores and reduce energy consumption while keeping the high performance.

Once Algorithm 1 returns the k-tuple, the adjuster tunes the frequencies of the cores accordingly using DVFS. After DVFS, the cores are organized into c-groups according to their frequencies. EEWA then allocates task classes to these c-groups accordingly. Assuming a_i $(0 \leq i \leq k-1)$ is an element in the k-tuple, task class TC_i is then allocated to the c-group of cores operating at frequency F_{a_i}.

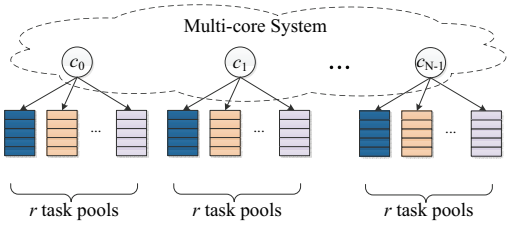
It is worth noting that the overhead of our backtracking algorithm is not large since the CC table is usually small. This is because cores in modern multi-core processors can only run at a few fixed frequencies (e.g. 4 different frequencies) and there are not many task classes in our parallel applications. As the backtracking algorithm does not use the number of cores but use the number of available frequencies, it is scalable. Our experimental results in Section IV show that the overhead of the backtracking algorithm is negligible for the applications on our multi-core machine.

B. Preference-based task-stealing scheduler

After adjusting core frequencies, EEWA adopts the preference-based task-stealing scheduler to schedule tasks in the next iteration in an energy-efficient manner.

As described before, the frequency adjuster allocates task classes to c-groups. Each c-group has a task pool to store the tasks allocated to it. Though using a centralized task pool is easy to implement, its serious lock contention can degrade the system performance. Therefore, we have adopted distributed task pools with the task-stealing policy, where a task pool is a double-ended queue which is convenient for task stealing.

Task-stealing relieves the lock contention of the task pools by using an individual task pool for each core. Most often a core obtains tasks from its local task pool without locking. Only when its local task pool is empty, it tries to steal tasks from other cores using locking. Since there are multiple task pools for stealing, the lock contention is usually low.



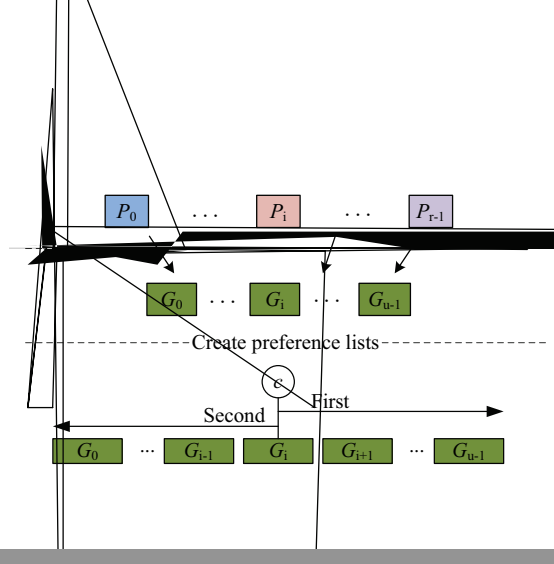


Figure 6. Normal
marks in Cilk, Cil

The figure s

The low ener

frequency adjuster, the cores are adjusted to appropriate frequencies according to the workload information of tasks, and then the tasks are allocated to the proper c-groups according to k-tuple. Together with the preference-based task scheduler, the cores can finish their tasks within the same time as they run at the fastest speed, but with less energy due to lower frequencies set for the cores.

The high performance of the applications in EEWA has been maintained due to the well balanced workloads among cores. With the preference-based task scheduler, even the workloads are not balanced as expected due to the variation of the workloads of the tasks in different batches, EEWA can adjust the workloads among different cores at runtime.

Compared to Cilk, Cilk-D can also reduce energy consumption of the benchmarks in multi-core systems with DVFS support. This is because free cores are scaled down to the lowest frequency if all the task pools are empty in Cilk-D. As a result, Cilk-D can reduce energy consumption of benchmarks ranging from 6.7% to 12.8% compared to Cilk. However, because Cilk-D is not aware of the workloads of the tasks, the cores cannot finish the tasks at the same time. In this situation, all the cores need to wait for the last core to arrive at a barrier. These cores waste extra energy and therefore the applications consume more energy in Cilk-D than in EEWA.

WATS [9] is a near-optimal work-stealing scheduler that proposes the *rob-the-weak-first* principle for asymmetric multi-core systems. Fig. 7 shows the performance of benchmarks in EEWA, WATS and Cilk on asymmetric multi-core systems in which the frequencies of cores are configured by EEWA. For each benchmark, the frequencies of cores are configured as the most often used frequency configurations in different batches of the benchmark (see Fig. 8 later).

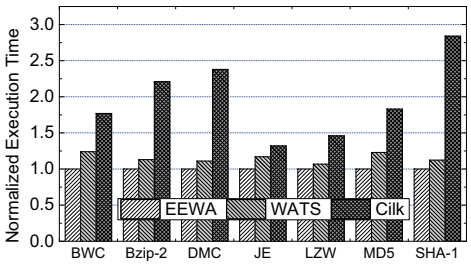


Figure 7. Performance of benchmarks on asymmetric multi-core systems in which the frequencies of cores are configured by EEWA.

As shown in Fig. 7, Cilk degrades the performance of benchmarks seriously because the random work-stealing policy in Cilk may schedule tasks with heavy workloads to slow cores. The execution times of the benchmarks in Cilk are about 1.17 to 2.92 times of their execution times in EEWA while their execution times in WATS are about 1.05 to 1.24 times of their execution times in EEWA.

Compared to Cilk, WATS improves the performance of benchmarks because WATS schedules tasks with heavy workloads to fast cores. However, even the frequencies of cores are configured by EEWA, the performance of WATS is still worse than the performance of EEWA because WATS cannot adjust the frequencies of cores to the optimal according to the workloads of benchmarks in different batches.

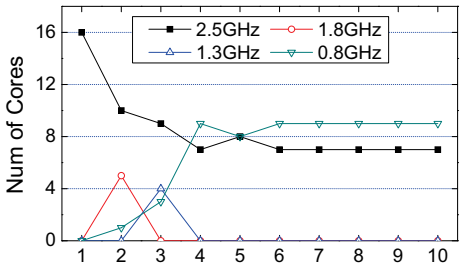


Figure 8. Number of cores with four frequencies in the 10 batches of SHA-1 respectively.

Fig. 8 shows the number of cores operating at each of the four frequencies in the 10 batches of SHA-1 with EEWA. During the first batch, all the cores run at the highest frequency (i.e., 2.5GHz in our platform). Then, at the end of each batch, the workload-aware frequency adjuster tunes the frequencies of cores according to the optimal frequency configuration found by the backtracking algorithm. As shown in Fig. 8, except the first batch, more than half of the cores are adjusted to run at the lowest frequency in most batches. For example, from the 3rd to the 10th batch, 5 cores run at 2.5GHz but the other 11 cores run at 0.8GHz.

B. Overhead of EEWA

For all the benchmarks in Table II, Table III gives their execution time and the overhead of the backtracking algorithm for searching appropriate k-tuple (Algorithm 1) in EEWA in millisecond (ms).

Observing Table III, we can find that the overhead of EEWA only costs less than 2% of the overall execution time for all the benchmarks. Therefore, the overhead of EEWA is negligible for the applications in our experiment.

Table III
THE EXECUTION TIME AND THE OVERHEAD OF EEWA IN MILLISECOND

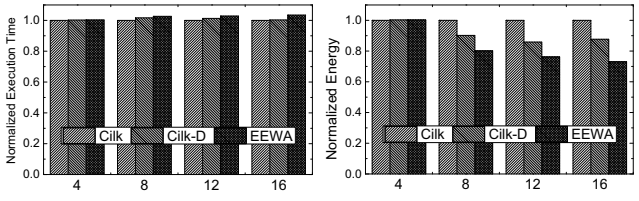
Benchmarks	Execution Time	Overhead	Percentage of overhead
BWC	16505	40.3	0.24%
Bzip-2	1907	37.5	1.97%
DMC	28413	14.9	0.05%
JE	3469	12.7	0.37%
LZW	4981	48.9	0.98%
MD5	6293	33	0.52%
SHA-1	3334	41.8	1.25%

Note that, the overhead of EEWA does not change dramatically in different applications. They are less than 100ms. To

keep the extra overhead low, EEWA favors applications that run for at least two seconds so that the overhead of EEWA will be smaller than 5% of the overall execution time.

C. Scalability of EEWA

Fig. 9 shows the scalability of EEWA. It gives the execution time and the energy consumption of DMC in Cilk, Cilk-D and EEWA on multi-core architectures with 4 cores, 8 cores, 12 cores and 16 cores. Other benchmarks show similar results.



(a) Performance of DMC (b) Energy consumption of DMC

Figure 9. Normalized execution time and energy consumption of DMC in Cilk, Cilk-D and EEWA on multi-core systems with different number of cores.

In the figure, the x-axis represents the number of cores used by the application. From the figure we can find that EEWA works very well in different hardware configurations. When the number of cores is small (e.g., 4 cores), all the cores are adjusted to run at the highest frequency since there are many tasks to be executed by the cores in each batch. In this case, the energy consumption cannot be reduced at all but the performance only degrades slightly (0.3%) for DMC. This result further shows that the overhead of EEWA is negligible. On the other aspect, EEWA can significantly reduce the energy consumption compared with both Cilk and Cilk-D if there are many cores (e.g., 16 cores). For example, when there are 12 cores, EEWA reduces 23.8% energy consumption but only degrades performance of DMC by 2.8% compared with Cilk.

From the figure we can also find that EEWA can reduce more energy consumption when the number of cores increases. When the number of cores is large, in most cases, the tasks cannot fully utilize all the available computational capacity. By scaling down the frequencies of some cores, EEWA can reduce the energy consumption while the hardware can still provide the required performance. Therefore, we can conclude that EEWA works more effectively with the increasing of the number of cores. This feature is very promising for future multi-core processors since hardware manufacturers keep producing CPU chips with more cores.

D. Discussion

EEWA aims at reducing the energy consumption of task-based CPU-bound applications. For memory-bound applications, because their execution time does not have a simple model related to CPU frequencies according to our experiment, currently EEWA cannot build the CC table for memory-bound applications within a reasonable time. Therefore, EEWA has to decide at runtime if an application is CPU-bound or memory-bound based on profiled information. When EEWA collects the execution time of every task in the first batch, it also collects the cache misses and the number of retired instructions of the task through performance monitoring counter. If the *cache miss intensity* (i.e., cache misses per instruction) of a task is larger than a given threshold, the task is labelled as memory-bound. If most tasks of an application are memory-bound, the application is regarded as memory-bound by EEWA. Currently, for memory-bound applications, EEWA simply adopts the traditional work-stealing for the rest of the batches. As a future work, we will extend EEWA for memory-bound applications using machine learning techniques. By analyzing the execution time of memory-bound tasks on cores of different frequencies through machine learning, it is possible for EEWA to create a correct CC table for memory-bound applications. Once the CC table is built, EEWA can reduce energy consumption for memory-bound applications with slight performance loss. Techniques in [11], [12] can also be integrated into EEWA to improve the performance of memory-bound applications in future.

The general ideas of EEWA can also be adapted to reduce energy consumption of parallel applications that do not launch tasks in batches, although we propose EEWA for batch-based applications. For a parallel application that does not launch tasks in batches, we can collect the workload information of the tasks by profiling the application offline. Once the information is collected, we can use the workload-aware frequency adjuster and the preference-based task scheduler to improve the energy efficiency of the application in the later executions.

V. Related work

Due to energy crisis and global warming, energy efficiency has become a popular research topic. Existing studies on energy efficiency generally have two important directions. One is how to maximize performance with a given energy budget. The other is how to reduce energy consumption without degrading performance conspicuously.

The first research direction is rooted in the heat dissipation problem in multi-core processors where more and more cores are integrated into a single CPU chip. A solution to the heat dissipation problem is to set an energy budget to the CPU chip [16]. With a given energy budget, many researches have been done to maximize performance of parallel applications [26], [28], [8]. In [28], a hierarchical and a gradient ascent-based technique is proposed for decentralized peak power management while other techniques use centralized decision making that cannot prevent instantaneous power from exceeding the peak power budget. The technique prevents power from exceeding the peak power budget.

In [8], to save energy, low-confident speculative instructions that consume a lot of energy in the instruction pipeline (about 30% of the overall energy [25]) are reduced and the non-critical instructions that are in cycles exceeding the energy budget are delayed. With a given energy budget, *Power Token Balancing* (PTB) [1] improves the overall performance of a parallel program by using the to-be-saved energy from non-critical threads for speeding the critical threads (i.e., threads that decide the timespan of the program). In this way, PTB can reduce the execution time of critical threads and hence can improve the overall performance. In [23], a scalable power control solution is proposed to maximize the performance of many-core processors with a given energy budget.

The second research direction attempts to save energy without degrading the performance. EEWA follows this research direction. Currently, most of the studies target parallel programs with threads (similar to the tasks in EEWA) that run the same code and synchronize at the same barrier [20], [21], [30], [22], [7], [3]. For example, [20] proposes a *thrifty barrier*, where threads that arrive at the barrier earlier adjust their cores to a low power mode to save energy. When the last thread (aka., critical thread) arrives at the barrier, all the other threads are woken up from the low power mode. Apart from the thrifty barrier, many other studies adjust the frequencies of cores so that all the threads can arrive at the barrier at the same time [21], [7]. In these studies, the last thread to arrive at the barrier (aka. critical thread) has to be identified in advance. Once the critical thread is identified, the cores that execute non-critical threads are adjusted to run at lower frequencies accordingly. Many studies have proposed algorithms identifying the critical thread from the threads that synchronize at the same barrier. In [21], a history-based method is proposed. In [7], a *meeting point* is proposed based on an extra counter. It also proposes *thread delaying* to adjust the frequencies of cores once the identified thread is identified. In [3], a *thread criticality predictor* (TCP) is proposed at the hardware level.

The aforementioned techniques assume that each core can only execute one thread at the same time. However, in modern multi-core processors with simultaneous multi-threading (SMT) support, each core can execute multiple threads at the same time. On these multi-core processors, if multiple threads on the same core require the core to adjust to different frequencies at the same time, all the previous techniques would not work. To solve the problem, *thread shuffling* [6] migrates threads that suggest similar frequencies to the same core.

Different from the above assumption that threads should execute the same code, EEWA allows tasks that synchronize at the same barrier to run different code sets and to have different workloads. All the existing techniques cannot handle this situation since they cannot identify the critical tasks running different code sets. To the best of our knowledge, EEWA is the first task scheduler that targets parallel programs with tasks have different workloads.

The central issue in scheduling a parallel program in energy-efficient manner is how to find the optimal frequency configuration for the hardware platform so that the program can save energy but keeps the same performance. To address this issue, in [29], in so-energy-efficiency model has been proposed to predict energy consumption when a parallel system scales up. In [14], several distributed Dynamic Voltage Scaling (DVS) scheduling schemes are implemented and applied to power-aware clusters. While these studies mainly focus on the complex, theoretical performance-energy models, our workload-aware frequency adjuster in EEWA uses a simple backtracking algorithm to search for the optimal/near-optimal frequency configuration at runtime.

The *rob-the-weaker-first* principle is also used in WATS [9], [10] that is proposed to improve the performance of parallel applications with different workloads on asymmetric multi-core architectures. In WATS, the preference lists of cores do not change since the frequencies of all the cores do not change at all. However, the preference lists of cores in our preference-based task-stealing scheduler may change in different iterations because EEWA may use different task pools in different iterations. Therefore, compared with WATS, EEWA uses the *rob-the-weaker-first* principle in a more complex context.

VI. Conclusions and Future works

Current parallel programming environments and task schedulers do not consider DVFS that can be used to reduce energy consumption in modern multi-core architectures. To reduce energy consumption without degrading performance for parallel applications, we have smartly used DVFS and online profiling in the energy-efficient EEWA task scheduler. EEWA uses a workload-aware frequency adjuster to tune the frequencies of the cores according to the workload information of tasks collected with online profiling. Besides the frequency adjuster, EEWA adopts a preference-based task-stealing algorithm to balance workloads among cores dynamically for high performance. Experimental results demonstrate that EEWA can reduce energy consumption up to 29.8% with a slight performance degradation for CPU-bound applications.

Acknowledgment

This work was partially supported by Shanghai Excellent Academic Leaders Plan (No. 11XD1402900), Program for Changjiang Scholars and Innovative Research Team in University (IRT1158, PCSIRT) China, NSFC (Grant No. 61272099, 61261160502) and Scientific Innovation Act of STCSM (No.13511504200). This work was also partially supported by JSPS Research Fellowships for Young Scientists Program.

REFERENCES

[1] J. Aragón, S. Kaxiras, et al. Power token balancing: Adapting CMPs to power constraints for parallel multithreaded workloads. In *IPDPS*, pages 431–442. IEEE, 2011.

[2] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of openmp tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.

[3] A. Bhattacharjee and M. Martonosi. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. *ACM SIGARCH Computer Architecture News*, 37(3):290–301, 2009.

[4] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *The Journal of Parallel and Distributed Computing*, 37(1):55–69, Aug. 1996.

[6] Q. Cai, J. González, G. Magklis, P. Chaparro, and A. González. Thread shuffling: Combining dvfs and thread migration to reduce energy consumptions for multi-core systems. In *International Symposium on Low Power Electronics and Design*, pages 379–384. IEEE, 2011.

[7] Q. Cai, J. González, R. Rakvic, G. Magklis, P. Chaparro, and A. González. Meeting points: using thread criticality to adapt multicore hardware to parallel regions. In *PACT*, pages 240–249. ACM, 2008.

[8] J. Cebrián, J. Aragón, J. García, P. Petoumenos, and S. Kaxiras. Efficient microarchitecture policies for accurately adapting to power constraints. In *IPDPS*, pages 1–12. IEEE, 2009.

[9] Q. Chen, Y. Chen, Z. Huang, and M. Guo. WATS: Workload-Aware Task Scheduling in asymmetric multi-core architectures. In *IPDPS*, pages 249–260. IEEE, 2012.

[10] Q. Chen and M. Guo. Adaptive workload aware task scheduling for single-ISA multi-core architectures. *ACM Transactions on Architecture and Code Optimization (to appear)*, 2014.

[11] Q. Chen, M. Guo, and Z. Huang. CATS: Cache Aware Task-Stealing based on Online Profiling in Multi-socket Multi-core Architectures. In *ICS*, pages 163–172. IEEE, 2012.

[12] Q. Chen, M. Guo, and Z. Huang. Adaptive cache aware bi-tier work-stealing in multi-socket multi-core architectures. *IEEE Transactions on Parallel and Distributed System*, 24(12):2334–2343, 2013.

[13] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. *ACM Sigplan Notices*, 33(5):212–223, 1998.

[14] R. Ge, X. Feng, and K. Cameron. Performance-constrained distributed DVS scheduling for scientific applications on power-aware clusters. In *SC*, pages 34–44. IEEE, 2005.

[15] M. Hill and M. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[16] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi. An analysis of efficient multi-core global power management policies: maximizing performance for a given power budget. In *MICRO*, pages 347–358. IEEE, 2006.

[17] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *ISCA*. IEEE, 2004.

[18] J. Lee and J. Palsberg. Featherweight X10: a core calculus for async-finish parallelism. In *PPoPP*, pages 25–36. ACM, 2010.

[19] C. E. Leiserson. The cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[20] J. Li, J. Martinez, and M. Huang. The thrifty barrier: energy-aware synchronization in shared-memory multiprocessors. In *HPCA*, pages 14–23. IEEE, 2004.

[21] C. Liu, A. Sivasubramaniam, M. Kandemir, and M. Irwin. Exploiting barriers to optimize power consumption of CMPs. In *IPDPS*, pages 5a–5a. IEEE, 2005.

[22] Y. Luo, V. Packirisamy, W. Hsu, and A. Zhai. Energy efficient speculative threads: dynamic thread allocation in Same-ISA heterogeneous multicore systems. In *PACT*, pages 453–464. ACM, 2010.

[23] K. Ma, X. Li, M. Chen, and X. Wang. Scalable power control for many-core architectures running multi-threaded applications. In *ISCA*, pages 449–460. ACM, 2011.

[24] P. Macken, M. Degrauwe, M. Van Paemel, and H. Oguey. A voltage reduction technique for digital systems. In *International Solid-State Circuits Conference, Digest of Technical Papers*, pages 238–239. IEEE, 1990.

[25] S. Manne, A. Klauser, and D. Grunwald. Pipeline gating: speculation control for energy reduction. *ACM SIGARCH Computer Architecture News*, 26(3):132–141, 1998.

[26] K. Meng, R. Joseph, R. Dick, and L. Shang. Multi-optimization power management for chip multiprocessors. In *PACT*, pages 177–186. ACM, 2008.

[27] J. Reinders. *Intel threading building blocks*. O'Reilly, 2007.

[28] J. Sartori and R. Kumar. Distributed peak power management for many-core architectures. In *Design, Automation and Test in Europe*, pages 1556–1559. IEEE, 2009.

[29] S. Song, C. Su, R. Ge, A. Vishnu, and K. Cameron. Iso-energy-efficiency: an approach to power-constrained parallel computation. In *IPDPS*, pages 128–139. IEEE, 2011.

[30] C. Yu and P. Petrov. Low-cost and energy-efficient distributed synchronization for embedded multiprocessors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 18(8):1257–1261, 2010.