WATS: Workload-Aware Task Scheduling in Asymmetric Multi-core Architectures

Quan Chen* , Yawen Chen†, Zhiyi Huang†, Minyi Guo*

*Department of Computer Science, Shanghai Jiao Tong University, Shanghai, China
chen-quan@sjtu.edu.cn, guo-my@cs.sjtu.edu.cn

†Department of Computer Science, University of Otago, New Zealand
yawen@cs.otago.ac.nz, hzy@cs.otago.ac.nz

***Abstract*—Asymmetric Multi-Core (AMC) architectures have shown high performance as well as power efficiency. However, current parallel programming environments do not perform well on AMC due to their assumption that all cores are symmetric and provide equal performance. Their random task scheduling policies, such as task-stealing, can result in unbalanced workloads in AMC and severely degrade the performance of parallel applications. To balance the workloads of parallel applications in AMC, this paper proposes a Workload-Aware Task Scheduling (WATS) scheme that adopts history-based task allocation and preference-based task stealing. The history-based task allocation is based on a near-optimal, static task allocation using the historical statistics collected during the execution of a parallel application. The preference-based task stealing, which steals tasks based on a preference list, can dynamically adjust the workloads in AMC if the task allocation is less optimal due to approximation in the history-based task allocation. Experimental results show that WATS can improve the performance of CPU-bound applications up to 64% compared with the random task scheduling policies.**

***Keywords*-Workload-aware, Asymmetric Multicore architectures, Load balancing, Task scheduling, Task-stealing**

I. Introduction

Multi-core processors have become mainstream since they have better performance per watt and larger computational capacity than complex single-core processors. While chip manufacturers like AMD and Intel keep producing new CPU chips with more symmetric cores, researchers are investigating alternative multi-core organizations such as Asymmetric Multi-Core (AMC) architectures, where individual cores have different computational capabilities [1], [2], [3], [4].

AMC is attractive because it has the potential to improve system performance, to reduce power consumption, and to mitigate Amdahl's law [1], [4]. Since an AMC architecture consists of a mix of fast cores and slow cores, it can better cater for applications with a heterogeneous mix of workloads [2], [3]. For example, fast, complex cores can be used to execute the serial code sections, while slow, simple cores can be used to crunch numbers in parallel, which is more power-efficient. Many modern multi-core chips offer Dynamic Voltage and Frequency Scaling (DVFS) which can dynamically adjust the operating frequency of each core and thus is able to turn a symmetric multi-core chip into a performance-asymmetric multi-core chip.

Despite the rapid development of the AMC technology, current parallel programming environments, as listed below, still assume all cores provide equal performance. Due to this assumption, parallel applications cannot utilize the asymmetric cores of an AMC architecture effectively.

Most current parallel programming environments adopt either task-sharing or task-stealing (aka. work-stealing) policies for task scheduling. For example, MIT Cilk [5], Cilk++ [6], TBB [7], Java's fork-join framework [8], and X10 [9] adopt task-stealing, while OpenMP [10] uses task-sharing. Task-stealing is increasingly popular due to its good scalability and high performance [11].

However, both task-stealing and task-sharing allocate tasks randomly to different cores, which is not a problem for symmetric cores but can cause extremely unbalanced workloads among asymmetric cores. For example, a long task may be scheduled to a slow core, while a short task is executed by a fast core. This problem of unbalanced workloads, which will be further discussed in detail in Section II, can severely degrade the performance of parallel applications. To the best of our knowledge, no study has addressed this problem and investigated the optimal task scheduling in parallel programming environments so that applications that are comprised of parallel tasks with different workloads can perform efficiently in AMC.

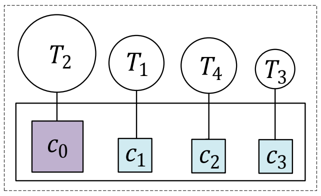
The rest of this paper is organized as follows. Section II describes the problem of unbalanced workloads in AMC and its solutions. Section III presents the WATS scheme that adopts a history-based task allocation algorithm and a preference-based task-stealing policy. Section IV provides experimental results, performance evaluation, and limitations of WATS. Section V discusses related work. Section VI summarizes our contributions, draws conclusions and sheds light on future work.

* Quan Chen was an assistant research fellow at the University of Otago during the course of this research.

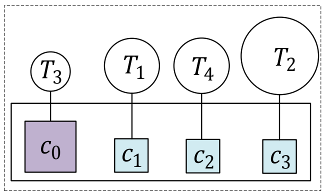
II. Motivation

A. The Problem

Let us use an example to explain the problem of unbalanced workloads in AMC. Suppose a parallel application has four parallel tasks: T_1, T_2, T_3 and T_4. We assume the application runs on an AMC architecture as shown in Fig. 1, with one fast core (c_0) and three slow cores (c_1, c_2 and c_3), where c_1, c_2 and c_3 have the same speed but the speed of c_0 is twice the speed of the slow cores. Note the speed here can be more precisely represented by the operating frequency of the cores. Suppose T_1, T_2, T_3 and T_4 take times $1.5t$, $4t$, t and $1.5t$ on the fast core c_0 respectively and all the tasks are CPU-bound, then we can reasonably deduce that T_1, T_2 T_3 and T_4 would take $3t$, $8t$, $2t$ and $3t$ on the slow cores.



(a) Optimal allocation



(b) A possible random allocation

Figure 1. Two possible allocations of T_1, T_2, T_3 and T_4.

Fig. 1 shows two possible allocations of T_1, T_2, T_3 and T_4 to the cores. Fig. 1(a) is an optimal allocation where T_2 is allocated to the fast core c_0 and the shorter tasks are allocated to the slow cores. The makespan (i.e., the overall completion time) for T_1, T_2, T_3 and T_4 is $max\{3t, 4t, 2t, 3t\} = 4t$.

However, with random scheduling policies such as task-stealing, T_1, T_2, T_3 and T_4 are likely to be randomly allocated as in Fig. 1(b), where T_3 is allocated to the fast core but the long task T_2 is scheduled to a slow core. In this case, the makespan for T_1, T_2, T_3 and T_4 is $max\{3t, 8t, t, 3t\} = 8t$. Obviously, allocating a long task to a slow core can degrade the overall performance seriously.

Some studies (e.g., [12]) tried to improve the random scheduling on AMC by allowing idle fast cores to snatch tasks from slow cores. For example, with this rescuing policy, for the situation in Fig. 1(b), c_0 is allowed to snatch T_2 from c_3 after finishing T_3. Suppose c_0 snatches T_2 from c_3 after finishing T_3 (which takes time t). c_0 still needs $(1 - \frac{t}{8t}) \times 4t = 3.5t$ to finish T_2 because c_3 has only finished $\frac{t}{8t}$ of T_2. Let Δ_s represent the time of the snatching operation. Then the overall time for c_0 to finish both T_3 and T_2 is $t + 3.5t + \Delta_s = 4.5t + \Delta_s$. Therefore, with the snatching policy, the makespan for T_1, T_2, T_3 and T_4 is $max\{3t, 4.5t + \Delta_s, t, 3t\} = 4.5t + \Delta_s$. If the system knows the workload of each task and Δ_s is not too large, the snatching policy can improve the performance of random scheduling, though it is still far from the optimal allocation.

However, since the workloads of the tasks are unknown to the existing random schedulers, idle fast cores have to snatch tasks randomly and thus the snatching policy will still suffer from the randomness in the random scheduling. For example, in Fig. 1(b), with the random snatching, the worst case could be that c_0 first snatches T_1 and T_4 before snatching T_2, where the makespan is roughly $5.25t + 3\Delta_s$.

In summary, the knowledge of task workloads is essential to optimal task scheduling in AMC. This knowledge can help a scheduler allocate long tasks to fast cores, which is often optimal. It can also help idle fast cores to steal or snatch the long tasks if steal and snatch are necessary. It is worth noting that an initial optimal allocation based on the knowledge of workloads is more crucial to the makespan than the snatching policy that tries to rescue a non-optimal allocation.

In the rest of this section, we will generalize the task allocation problem, assuming the workloads of tasks are known. We will give theoretical analysis on the optimal task allocation, which will guide our design and implementation of task scheduling in AMC.

Fig. 2 illustrates the general problem of task allocation. Suppose there are m independent tasks (γ_1, ..., γ_m) with different workloads and an AMC with cores operating at k different speeds (or frequencies) in descending order: F_1, ..., F_k. The number of cores operating at F_i is N_i ($1 \leq i \leq k$). The problem is to divide the m tasks into k groups that are assigned to the k core groups (denoted as c-groups) respectively, so that the makespan of the m tasks is minimum. In the problem, we assume that tasks can be optimally or near-optimally scheduled with random scheduling policies inside the same c-group with symmetric cores, which is a valid assumption for the task-stealing policy [11].

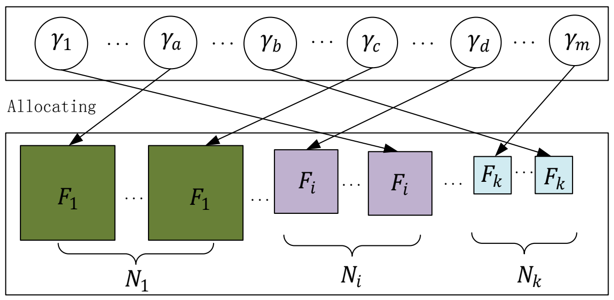


Figure 2. The task allocation problem in AMC.

B. Ideal solution

The following lemma and theorem provide theoretical guidance to optimal allocation.

Lemma 1. *Given m tasks γ_1, γ_2, ..., γ_m, suppose the workload of each task $\gamma_j (1 \leq j \leq m)$ is w_j. The lower* *bound of the makespan for the m tasks to run on k c-groups, each of which has N_i cores with speed F_i $(1 \le i \le k)$, is*

$$T_L = \frac{\sum_{j=1}^{m} w_j}{\sum_{i=1}^{k} F_i \times N_i}.$$

Proof: We adopt proof by contradiction. Suppose there exists a makespan T_s, where $T_s < T_L$. Since the makespan T_s is the execution time of the last core that finishes its tasks, the overall workloads of tasks processed by all the cores must be smaller than $\sum_{i=1}^{k}(T_s \times F_i \times N_i) = T_s \cdot \sum_{i=1}^{k} F_i \times N_i < T_L \cdot \sum_{i=1}^{k} F_i \times N_i = \sum_{j=1}^{m} w_j$. This contradicts the fact that the overall workloads of all the tasks are $\sum_{j=1}^{m} w_j$. Therefore, there must be no makespan that is shorter than T_L.

Theorem 1. *For tasks γ_1, ..., γ_m, if $\gamma_{p_{i-1}+1}$, ..., γ_{p_i} $(1 \le i \le k,\ p_0 = 0,\ p_k = m)$ are allocated to the c-group with speed F_i, their makespan is T_L only when p_1, ..., p_{k-1} satisfy*

$$\begin{aligned}\sum_{j=1}^{p_1} w_j : ... : \sum_{j=p_{i-1}+1}^{p_i} w_j : ... : \sum_{j=p_{k-1}+1}^{m} w_j \\ = F_1 \times N_1 : ... : F_i \times N_i : ... : F_k \times N_k\end{aligned} \quad (1)$$

Moreover, the task allocation is optimal and the makespan is $\frac{\sum_{j=1}^{p_1} w_j}{F_1 \times N_1} = \frac{\sum_{j=p_{i-1}+1}^{p_i} w_j}{F_i \times N_i} ... = \frac{\sum_{j=p_{k-1}+1}^{m} w_j}{F_k \times N_k} = T_L$.

Proof: As can be seen from Lemma 1, if the makespan is T_L, all the cores are fully utilized and they complete tasks allocated to them at the same time. Thus the workloads of tasks that are allocated to the c-group with core speed F_i are $F_i \times N_i \times T_L$. Therefore, $\sum_{j=1}^{p_1} w_j : ... : \sum_{j=p_{i-1}+1}^{p_i} w_j : ... : \sum_{j=p_{k-1}+1}^{m} w_j = F_1 \times N_1 \times T_L : ... : F_i \times N_i \times T_L : ... : F_k \times N_k \times T_L = F_1 \times N_1 : ... : F_i \times N_i : ... : F_k \times N_k$.

If tasks are divided into groups in Eq. 1, the workloads are balanced among the k c-groups in terms of the computation capacities of the cores in different c-groups. Since all the workloads are fully balanced during the time period T_L and the lower bound is achieved, this task allocation is optimal. Therefore, the execution time for the group of tasks allocated on the k c-groups can be calculated as $\frac{\sum_{j=1}^{p_1} w_j}{F_1 \times N_1} = ... = \frac{\sum_{j=p_{i-1}+1}^{p_i} w_j}{F_i \times N_i} = ... = \frac{\sum_{j=p_{k-1}+1}^{m} w_j}{F_k \times N_k} = T_L$.

C. Proposed solution

However, it is not feasible to find the ideal solutions to Theorem 1 because they may not exist in real situations. Even if they exist, finding the solutions to Theorem 1 is similar to the *job shop scheduling* problem [13] which is NP-hard. The difference between Theorem 1 and the job shop scheduling problem is that the processors are symmetric in job shop scheduling problem while Theorem 1 assumes asymmetric cores.

Due to the above reasons, we relax the conditions of Theorem 1 and propose a near-optimal solution for the task allocation problem in AMC, as shown in Fig. 3.

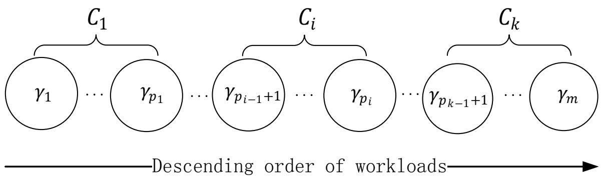


Figure 3. Allocate m tasks with different workloads to k c-groups.

In the solution, the m independent tasks are sorted in descending order of their workloads. Based on the sorted tasks, we choose p_1, ..., p_{k-1} to divide the m tasks into k groups that are allocated to the k c-groups (i.e., C_1, ..., C_k) according to Algorithm 1. In the solution, we assume there are enough tasks to be allocated to the c-groups. Algorithm 1 chooses p_i that satisfies $\frac{\sum_{j=p_{i-1}+1}^{p_i} w_j}{F_i \times N_i} \le T_L$ and $\frac{\sum_{j=p_{i-1}+1}^{p_i+1} w_j}{F_i \times N_i} > T_L$. In this way, we can keep $max(|\frac{\sum_{j=1}^{p_1} w_j}{F_1 \times N_1} - T_L|, ..., |\frac{\sum_{j=p_{k-1}+1}^{m} w_j}{F_k \times N_k} - T_L|)$ as small as possible.

Algorithm 1 Static near-optimal task allocation algorithm

```
Input:  A set of tasks $\{\gamma_1, ..., \gamma_m\}$.
Input:  The workload of $\gamma_i$ is $w_i$.
Input:  The speeds of the $k$ c-groups are $\{F_1, ..., F_k\}$
    (where $F_i > F_j$, if $i < j$).
Input:  The number of cores operating at $F_i$ is $N_i$.
1: compute the lower bound $T_L$ as in Lemma 1;
2: $w = 0$; $j = 1$; $i = 1$;
3: while $i \le m$ && $j \le k - 1$ do
4:     $w = w + w_i$;
5:     if $w > T_L \times N_j \times F_j$ then
6:         $p_j = i - 1$; $j$++; $w = w_i$;
7:     end if
8:     $i$++;
9: end while
Output:  $\{p_1, ..., p_{k-1}\}$.
```

In the above near-optimal solution, we assume the number of tasks and their workloads are known. However, in real parallel applications, this assumption is not valid because the tasks are generated dynamically and their workloads are not known until they are completed. How to apply the above theoretical solution to parallel programming environments is a challenging issue in our study.

In our implementation, we propose *history-based task allocation* to allocate a new task to the right c-group. Basically we use history to predict the pattern of future task workloads. Tasks are classified into task classes according to their function names. Instead of allocating dynamically generated tasks, we allocate the task classes to different c-groups. For the same function f, we can collect the average workload of the f-named tasks in the history. Since the number of different functions and their average workloads are known from history, we can adopt Algorithm 1 to allocate the functions to different c-groups. Based on this allocation, any newly generated task will be allocated to the c-group where its function name is allocated. If history can reasonably reflect future patterns of task generation, this task allocation scheme will work well.

We should admit there are times history may mis-predict the future. The above *history-based task allocation* is only an approximation of the optimal allocation. In order to further balance the workload, we propose a *preference-based task-stealing* policy to adjust the workloads dynamically among different c-groups. Each core is given a *preference list* of task clusters (to be defined shortly). An idle core steals a task according to the order of its preference list.

In the following section, we propose the Workload-Aware Task Scheduling (WATS) scheme, which adopts both the history-based task allocation algorithm and the preference-based task-stealing policy.

III. Workload-Aware Task Scheduling (WATS)

The philosophy behind WATS is based on our previous theoretical analysis: an optimal task allocation is more crucial to the makespan of parallel tasks than the rescuing policies like task snatching or stealing; and a workload-aware task snatching/stealing is better than random snatching/stealing. The history-based task allocation algorithm and the preference-based task-stealing policy are used to fulfill the philosophy.

Again, in the following discussion, without loss of generality, we assume the asymmetric cores in AMC are comprised of k c-groups C_1, ..., C_k, where C_i has N_i cores operating at speed F_i ($1 \leq i \leq k$), and $F_i > F_j$ if $i < j$.

A. History-based Task Allocation

There are two assumptions in this allocation algorithm. First, tasks executing the same function have similar workloads. Second, the percentage of tasks executing the same function among all tasks is almost the same during the execution of a parallel application. Based on the two assumptions, we use the historical statistics to guide the allocation of tasks to the k c-groups.

Tasks completed in history 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 the tasks completed, and w is the average workload of the tasks.

The workload of a task is measured with CPU cycles through a performance counter and normalized against the fastest core speed F_1. Suppose a task γ is completed by a core with speed F_i in n cycles, then γ's workload w_γ is calculated with Eq. 2.

$$w_\gamma = n \times \frac{F_i}{F_1} \tag{2}$$

Once a task γ is completed, the information of its task class $TC(f, n, w)$ is updated using Algorithm 2. If there is no such a class, a new task class is created for f.

Algorithm 2 Workload information updating algorithm

```
Input:  Completed task $\gamma$ with the function name $f$.
Input:  $\gamma$'s workload $w_\gamma$.
Input:  Existing task classes $\{TC_1, ..., TC_m\}$.
1: for each $TC_i(f_i, n_i, w_i) \in \{TC_1, ..., TC_m\}$ do
2:     if $f_i == f$ then
3:         $TC_i(f_i, n_i, w_i) \Rightarrow TC_i(f_i, n_i + 1, \frac{n_i \times w_i + w_\gamma}{n_i + 1})$;
4:         return ;
5:     end if
6: end for
7: create a new task class $TC_{m+1}(f, 1, w_\gamma)$;
```

Based on information about the task classes, the next step is to allocate the task classes to the k c-groups using Algorithm 1. We sort the task classes $TC_i(f_i, n_i, w_i)$ ($1 \leq i \leq m$) in descending order of w_i. Then we use the overall workload $n_i \times w_i$ as the workload of the task class $TC_i(f_i, n_i, w_i)$, when applying Algorithm 1, to divide the task classes into k groups and allocate them to the k c-groups accordingly. We call the k groups of task classes *task clusters*. Since task clusters and c-groups are a one-to-one mapping, for the sake of convenience, we use C_i to represent both a task cluster and a c-group in the following discussion.

With the above task clusters, we can allocate a newly generated task to a c-group in the following way. When a task γ with a function name f is generated, its task class is checked first. If the task class $TC(f, n, w)$ exists and belongs to the task cluster C_i, then γ is allocated to the c-group C_i. If there is no task class for f, then γ is allocated to the fastest c-group C_1 because we try to complete γ and collect the information of f's task class for future use as soon as possible.

It is worth noting that all the information used in the algorithm is collected automatically. The number of cores and their speeds can be acquired from the operating system. The number of CPU cycles of a task is acquired at runtime with a performance counter. Once a task is completed, the information about the task classes is updated and the task clusters are re-organized using Algorithm 1. Therefore, historical statistics are updated in a timely manner.

B. Preference-based Task-stealing

The above allocation algorithm divides tasks into k task clusters that are allocated to the k c-groups accordingly. Each c-group needs a task pool, which is a double-ended queue (aka. deque), to store the tasks allocated to it. Though using a centralized task pool is an easy technique for implementation, its serious lock contention can degrade the system performance. Therefore, we have adopted distributed task pools with the task-stealing policy.

Task-stealing can relieve the lock contention of the task pools. It provides an individual task pool for each core. Most often a core obtains tasks from its own task pool without locking. Only when a core's task pool is empty, should it try to steal tasks from other cores with locking. Since there are multiple task pools for stealing, the lock contention is much lower.

In our situation, task-stealing becomes more complex since each core needs k local task pools, labeled as C_1, ... C_i, ..., C_k, corresponding to the k task clusters. When a new task is generated, it is pushed into one of the local pools using the history-based task allocation algorithm. A core from the c-group C_i usually obtains tasks locally from its task pool C_i which stores tasks allocated to its c-group. If the task pool C_i is empty, it steals randomly from the C_i pools of other cores, as the traditional task-stealing policy. However, when all C_i pools are empty, which means all tasks allocated to the c-group C_i are completed, we should allow the c-group to execute tasks allocated to other c-groups in order to balance the workloads among the c-groups. The complexity arises when deciding which pool of tasks to choose in this situation. The following preference-based task-stealing gives our solution.

In the preference-based task-stealing policy, each core is given a *preference list* of task clusters. The preference list of a core contains all the k task clusters that are ordered as detailed below.

For a core in the c-group C_i, its preference list is created as $\{C_i, C_{i+1}, ..., C_k, C_{i-1}, C_{i-2}, ..., C_1\}$ as shown in Fig. 4.

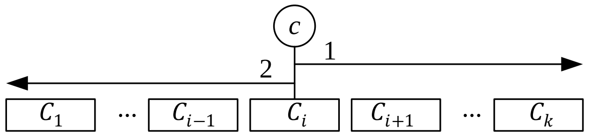


Figure 4. Preference list of the cores in the c-group C_i.

The preference list in Fig. 4 is generated based on the *rob the weaker first* principle. This principle can help reduce the makespan. For example, if a core steals a task that is allocated to faster cores, it needs a long time to execute the stolen task, which may prolong the makespan. On the contrary, if a core steals a task that is allocated to slower cores, it can execute the stolen task in a shorter time and relieve the pressure on slow cores. However, this preference list does not prevent slow cores to steal tasks from fast cores. When the slow cores have no tasks, they can steal tasks from the busy fast cores.

Algorithm 3 shows in detail the preference-based task-stealing policy adopted by each core.

Algorithm 3 Preference-based task-stealing

```
Input:  A core $c$ from the c-group $C_i$.
Input:  $c$'s preference list $\{C_i, ..., C_k, C_{i-1}..., C_1\}$.
 1: while $c$ has not obtained a task do
 2:   for each $C_j \in \{C_i, ..., C_k, C_{i-1}..., C_1\}$ do
 3:     $c$ tries to get a task from its local task pool $C_j$;
 4:     if succeed then
 5:       return ;
 6:     else
 7:       while there are some non-empty $C_j$ pools in other cores do
 8:         $c$ randomly chooses a victim core $v$;
 9:         $c$ steals a task from $v$'s task pool $C_j$;
10:         if succeed then
11:           return ;
12:         end if
13:       end while
14:     end if
15:   end for
16: end while
```

Fig. 5 shows an example architecture of WATS on an asymmetric quad-core architecture with cores operating at 3 different speeds. That is, there are three c-groups C_1 (with core c_0), C_2 (with c_1 and c_2) and C_3 (with c_3).

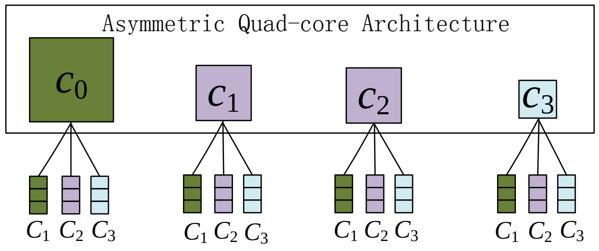


Figure 5. An example architecture of WATS. Each core adopts one task pool for each task cluster.

Therefore, tasks are classified into 3 task clusters (C_1, C_2 and C_3) and each core adopts 3 task pools C_1, C_2 and C_3 accordingly. The preference lists of the four cores are generated as in Table I based on the *rob the weaker first* principle as shown in Fig. 4. For example, c_3 will always look for tasks from the C_3 pools first, which have the tasks that are allocated to c_3's c-group using the history-based task allocation algorithm. Then it will search the C_2 pools, and finally the C_1 pools.

C. Implementation

WATS has been implemented in MIT Cilk. MIT Cilk is one of the earliest parallel programming environments that implement task-stealing [14]. It extends C with three keywords: *cilk*, *spawn* and *sync* to declare parallelism in the program. *cilk* identifies a procedure as a *Cilk procedure*,

Table I
PREFERENCE LISTS OF CORES

C-group	Core	Preference list
C_1	c_0	$\{C_1, C_2, C_3\}$
C_2	c_1 & c_2	$\{C_2, C_3, C_1\}$
C_3	c_3	$\{C_3, C_2, C_1\}$

spawn is used to generate a child task, and *sync* waits for all the child tasks that are spawned by the current task, to return.

MIT Cilk consists of a compiler and a scheduler. The Cilk compiler, named *cilk2c*, is a source-to-source translator that transforms a Cilk source into a C program. Once a task is generated, a task frame is created to store the information needed by the task and the scheduler. The Cilk scheduler uses the traditional task-stealing policy.

To help task classification, we have modified *cilk2c* to record a task's function name in the task frame. When a new task is spawned, it is subsumed into its task class according to its function name in the task frame. With the history-based allocation algorithm that groups the task classes into task clusters, WATS can allocate any new task to the corresponding task cluster.

WATS launches a helper thread to execute the history-based task allocation algorithm at runtime. The helper thread periodically (e.g., every 1ms) checks every core to find out if it has completed some tasks. Once there is a completed task, the helper thread updates the workload information of the task class with Algorithm 2 and re-organizes the task clusters with Algorithm 1. The helper thread is scheduled by the OS to any free core at runtime. Our experimental results show that the extra overhead incurred by the helper thread is very small.

Two types of task-generating policies, parent-first and child-first, can be adopted for task stealing. In the parent-first policy, a core continually executes the parent task after spawning a child task, leaving the child task for later execution or for stealing by other cores. One such example is the help-first policy proposed in [15]. In the child-first policy, however, a core executes the child task immediately after the child is spawned, leaving the parent task for later execution or for stealing by other cores. For example, the MIT Cilk uses the child-first policy, aka. work-first in [5].

WATS adopts the parent-first policy because it is difficult to collect the workload information of tasks with the child-first policy. If a core is executing a task γ, with the child-first policy, it is very likely the core will also execute γ's child tasks before γ is completed. Therefore, γ's workload information may not be collected correctly as it could include the workloads of γ's child tasks. As a result, we have modified *cilk2c* to spawn tasks with the parent-first policy.

IV. EVALUATION

We use a Dell 16-core computer that has four AMD Quad-core Opteron 8380 processors (codenamed "Shanghai") to evaluate the performance of WATS. In the processor, each core can run at 2.5GHz, 1.8GHz, 1.3GHz and 0.8GHz. We adjust the frequency of each core to emulate different AMC architectures. Table II lists the emulated AMC architectures in the experiment.

Table II
THE EMULATED AMC ARCHITECTURES

Freq. / Name	2.5 GHz	1.8 GHz	1.3 GHz	0.8 GHz
AMC 1	2	2	2	10
AMC 2	4	4	4	4
AMC 3	2	0	0	14
AMC 4	4	0	0	12
AMC 5	8	0	0	8
AMC 6	12	0	0	4

Since WATS is proposed to improve the performance of CPU-bound applications with tasks that have different workloads, the benchmarks in Table III are CPU-bound. The source code of benchmarks are from their official websites but adapted to run on MIT Cilk. In the batch-based benchmarks, the program launches many parallel tasks (e.g., 64 tasks) in each batch. When the tasks in one batch are completed, the program launches another batch of tasks. In the pipeline-based benchmarks, the execution of a program has several parallel stages. Tasks in different stages run in parallel but communicate with each other with pipelines.

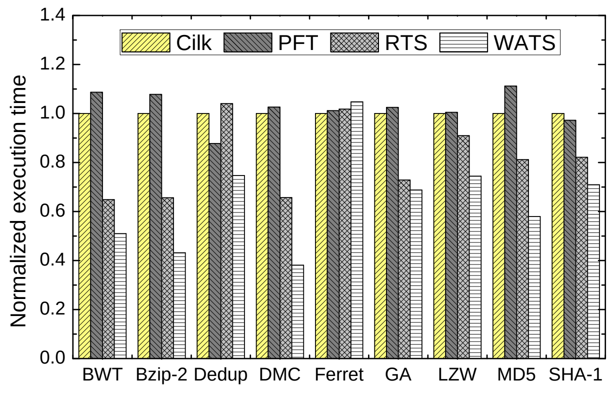
Table III
BENCHMARKS IN THE EXPERIMENT

Name	type	Description
BWT	Batch-based	Burrows Wheeler Transform
Bzip-2	Batch-based	Bzip2 file compression algorithm
DMC	Batch-based	Dynamic Markov Coding
GA	Batch-based	Island model of Genetic Algorithm
LZW	Batch-based	Lempel-Ziv-Welch data compression
MD5	Batch-based	Message Digest Algorithm
SHA-1	Batch-based	SHA-1 cryptographic hash function
Dedup	Pipeline-based	Dedup from PARSEC
Ferret	Pipeline-based	Ferret from PARSEC

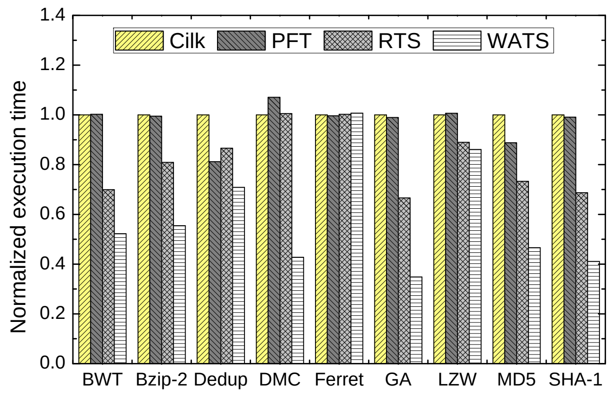
A. Performance of WATS

We compare the performance of WATS with the performance of three other task schedulers: MIT Cilk, PFT and RTS in AMC.

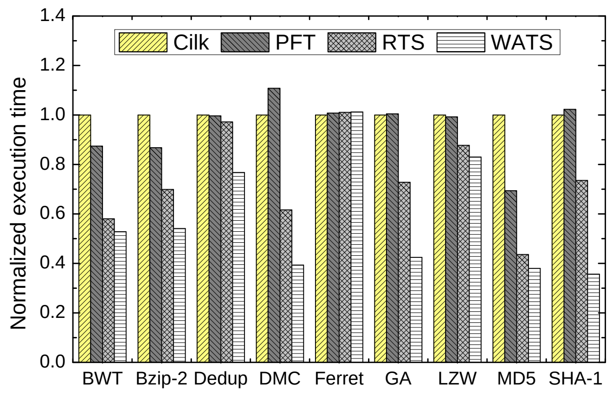
In MIT Cilk (denoted as Cilk for short) [5], tasks are spawned with the child-first policy and scheduled with the traditional task-stealing policy. In PFT (Parent-First Task-stealing) [15], tasks are spawned with the parent-first policy and scheduled with the traditional task-stealing policy. In RTS (Random Task-Snatching) [12], tasks are also spawned and scheduled as in Cilk, but a faster core snatches tasks from a randomly chosen slower core if the faster core cannot steal any task. To ensure fairness of comparison, WATS, PFT and RTS are implemented by modifying MIT Cilk.



(a) AMC 1



(b) AMC 2



(c) AMC 5

Figure 6. Performance of the benchmarks in AMC 1, AMC 2 and AMC 5.

We have tested the performance of the benchmarks in all the 6 AMC architectures shown in Table II. Fig. 6 only shows the performance of the benchmarks in AMC 1, AMC 2 and AMC 5 due to limited space, as the benchmarks in other AMC architectures perform similarly.

The figure shows that WATS can significantly improve the performance of the CPU-bound applications, with the performance gains ranging from 17% to 64% compared to Cilk and PFT, and with performance gains ranging from 6% to 57% compared to RTS. For example, for SHA-1 in Fig. 6(c), WATS reduces the execution time up to 64% compared to Cilk.

The good performance of WATS is due to its balanced workloads in the AMC architectures. With the history-based task allocation algorithm, WATS allocates tasks with heavy workload to fast cores and tasks with light workload to slow cores. Even if the workloads are not balanced as expected due to approximation, WATS can dynamically balance the workloads in AMC using the preference-based task-stealing policy.

On the contrary, in Cilk and PFT, it is very likely that tasks with heavy workload are scheduled to slow cores since tasks are stolen randomly. Scheduling a task with heavy workload to a slow core can seriously degrade the makespan of parallel tasks.

Compared to Cilk and PFT, RTS can slightly improve the performance of the benchmarks in AMC. This is because in RTS faster cores can randomly snatch tasks from slower cores and the snatched tasks can be completed earlier, which can reduce the makespan of the parallel tasks. As a result, comparing to Cilk and PFT, RTS improves the performance of the benchmarks ranging from 3% to 56%.

However, since RTS is not aware of the workloads of the tasks, it is possible for faster cores to snatch tasks with light workload, in which case the makespan cannot be reduced. Therefore, RTS still performs worse than WATS.

As shown in Fig. 6, the only benchmark of which WATS cannot improve the performance is *ferret*. This is because the parallel tasks in ferret have similar workloads and thus it is neutral to the history-based task allocation algorithm in WATS. However, the performance of this benchmark suggests that the extra overhead incurred by WATS is very small. As shown in Fig. 6(a), which is the worst case, the performance of ferret in WATS is only degraded by 4.7% compared to Cilk.

Fig. 7 shows the performance of the benchmark GA in all the 6 AMC architectures. From the figure, we can see that GA in WATS achieves better performance when an AMC architecture has more fast cores. For example, WATS reduces 31% execution time compared to Cilk and 19.5% execution time compared to RTS in AMC 3 (2 fast cores), while it reduces 59% execution time compared to Cilk and reduces 40.5% execution time compared to RTS in AMC 6 (12 fast cores).

Fig. 7 also shows WATS can adapt to different AMC architectures automatically and improve performance accordingly. From Fig. 7, we can see WATS can balance workloads adaptively in different AMC architectures. When there are more fast cores in an AMC architecture, WATS can

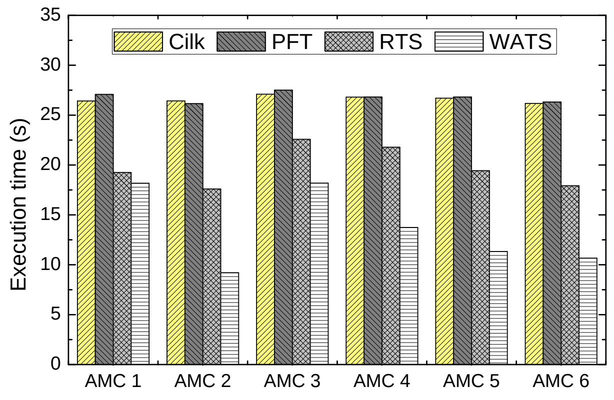


Figure 7. Performance of GA in all the 6 AMC architectures.

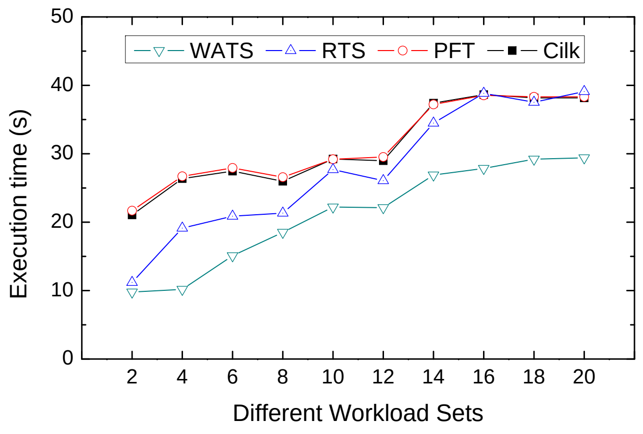


Figure 8. Performance of GA with different workloads in AMC 5.

allocate more tasks with heavy workload to fast cores using the history-based task allocation algorithm. Therefore, the performance of GA improves accordingly when the number of fast cores increases. However, in Cilk and PFT, tasks with heavy workload are still possible to be scheduled to slow cores even when there are many fast cores, which degrades the performance. As a result, the performance of GA in Cilk and PFT has not been improved at all even when the number of fast cores increases from AMC 4 to AMC 6, as shown in Fig. 7.

GA in RTS achieves a slightly better performance when an AMC architecture has more fast cores because the fast cores can snatch tasks less randomly from the slow cores when the number of slow cores becomes small. However, the performance gain from the rescuing of the non-optimal task allocation in RTS is still much less than that resulting from the near-optimal task allocation in WATS.

B. Scalability of the history-based task allocation

Fig. 8 shows the scalability of the history-based task allocation algorithm. It gives the performance of GA under different distributions of workloads in AMC 5, though other benchmarks show similar results in various AMC architectures. In the experiment, GA launches 64 tasks with 4 different workloads (in proportion of $8t$, $4t$, $2t$ and t) in each batch. The number of tasks with each type of workload is adjusted to evaluate the scalability of the history-based task allocation algorithm when the number of tasks with heavy workload increases. The distribution of workloads of $8t$, $4t$, $2t$ and t follows the pattern $\alpha, \alpha, \alpha, 64-3\alpha$, where α is adjusted as shown by the x-axis in Fig. 8.

From the figure we can see that the history-based task allocation algorithm works fine under different distributions of workloads. When α is small and the workloads are mostly light, WATS reduces the GA execution time by 53.5% compared to Cilk. When α is large and the workloads are mostly heavy, WATS can still reduce the execution time by 23% compared to Cilk.

However, RTS does not work well when the workloads are mostly heavy (e.g. α is 20), as it does not improve the performance at all compared to Cilk and PFT. This is because fast cores are not able to snatch all the heavy tasks that are allocated to the slow cores when there are too many heavy tasks. Moreover, the extra overhead incurred by the snatching operations even slightly degrades the overall performance. This result again supports our philosophy of WATS that an optimal task allocation is more important than rescuing policies such as task snatching.

C. Effectiveness of preference-based task-stealing

To evaluate the effectiveness of the preference-based task-stealing policy, we compare the performance of WATS with WATS-NP, a scheduler that adopts the history-based task allocation algorithm but its preference-based task-stealing is not allowed to steal tasks that are allocated to other c-groups. In this way, WATS-NP is able to show only the performance of the history-based task allocation algorithm.

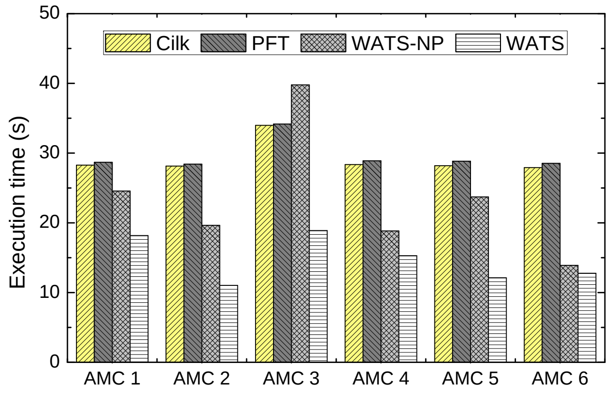


Figure 9. Performance of GA in Cilk, PFT, WATS and WATS-NP.

Fig. 9 shows the performance of GA in WATS and WATS-NP in all the 6 AMC architectures. From the figure we can see that the performance of WATS is always better than WATS-NP. The preference-based task-stealing in WATS is very helpful when handling slightly unbalanced workloads. Since the history-based task allocation algorithm may misallocate the tasks to the wrong c-groups due to its static approximation of the workloads of dynamic tasks, the preference-based task-stealing can remedy this imprecision. From Fig. 9 we can conclude that the preference-based task-stealing policy works effectively.

It is interesting to note that the history-based task allocation algorithm has mostly done effective allocation of tasks according to Fig. 9. Except for AMC 3, WATS-NP performs better than Cilk and PFT, which means the allocation algorithm is more effective than random task stealing in terms of load-balancing in AMC. As for the exception of AMC 3, it has only 2 fast cores but 14 slow cores. Therefore, an imprecision of the allocation algorithm can easily cause a large task to be allocated to slow cores with high probability when the number of slow cores is large. Fortunately, the preference-based task-stealing can remedy this imprecision effectively, as shown in the figure.

D. Task-snatching in WATS

It is of interest to discover whether or not task-snatching is also effective to WATS and thus should be integrated into WATS. To investigate this issue, we implemented a scheduler WATS-TS, where fast cores snatch tasks from slow cores when the fast cores cannot steal any tasks using the preference-based task-stealing policy.

In WATS-TS, when a core intends to snatch a task, it selects a slower core with the largest task. In this way, large tasks that affect the makespan seriously can be snatched to fast cores and completed earlier. Therefore, our workload-aware snatching policy is better than the *random snatching* in RTS, as explained in Section II-A. Moreover, workload-aware snatching causes fewer snatching operations than the random snatching, since randomly snatched small tasks take less time for the fast cores to complete, which causes the fast cores to snatch more often.

Fig. 10 shows the performance of all the benchmarks in WATS and WATS-TS in AMC 2. From the figure we see surprisingly that the performance of WATS-TS is slightly worse than WATS. Especially, for Bzip-2 and LZW, WATS-TS increases the execution time by 8% compared to WATS.

Fig. 10 tells us that WATS has satisfactorily balanced the workloads in AMC. When the workloads are balanced among cores in AMC, it is not worthwhile to snatch tasks from slower cores since the slower cores are also close to completion. The extra overhead incurred by the snatching operations simply makes WATS-TS perform worse. Therefore, there is no need for WATS to adopt the task-snatching policy.

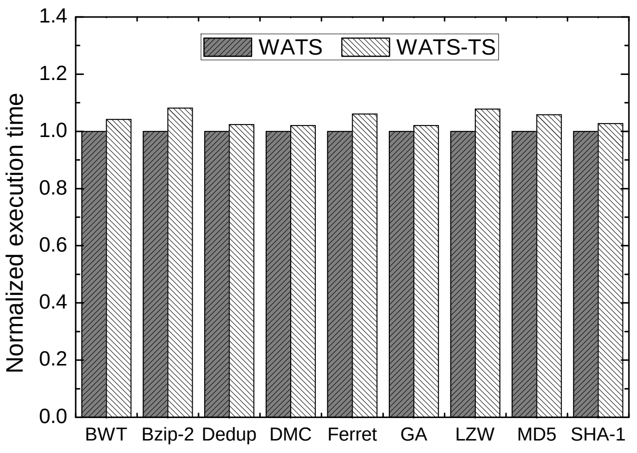


Figure 10. Performance of WATS and WATS-TS in AMC 2.

E. Discussion

Our experimental results have shown that WATS can significantly improve the performance of parallel applications in various AMC architectures. Both the history-based task allocation algorithm and the preference-based task-stealing policy in WATS have performed effectively, which nullifies extra optimizations such as task snatching in WATS.

WATS can be extended to work for applications with both CPU-bound and memory-bound tasks, though we have only presented the results for applications with CPU-bound tasks. We can decide if a task γ is CPU-bound or memory-bound in the following way. Given an AMC with k levels of caches and the cache miss penalty of the ith level cache is p_i. Let n_i represent the ith level cache misses of γ. The normalized cache misses of γ is $M = \sum_{i=1}^{k}(n_i \times \frac{p_i}{p_1})$. Suppose the number of instructions in γ is N, we can use CMPI (*Cache Misses Per Instruction*), $CMPI_\gamma = \frac{M}{N}$, to decide if γ is CPU-bound or memory-bound. If $CMPI_\gamma$ is greater than some threshold, γ is memory-bound and its performance depends on memory accessing time. We can allocate large CPU-bound tasks to fast cores, but allocate memory-bound tasks to slow cores because there will be no performance gain for memory-bound tasks to run on fast cores. The above information can be collected through performance counters at runtime. Furthermore, if most tasks are known to be memory-bound tasks from the initial stage, which simply indicates the application is memory-bound, WATS can easily adopt the random task-stealing for the rest of the execution. In this way, we can avoid the extra overhead of WATS, since it is indifferent for memory-bound tasks to run on a fast core or slow core and WATS is neutral to memory-bound applications.

Additionally, the above CMPI value can be used to save power in combination with DVFS. If the CMPI of a task is very large, we can scale down the operating frequency of the core using DVFS, because it has little impact on the performance of the task but saves power.

The general ideas in WATS can be applied to other situations. For example, WATS can be easily adapted to process-level scheduling in AMC if the processes are independent and their workloads can be estimated.

An interesting detail of the WATS implementation is that WATS schedules the main task of a parallel program on the fastest core. This is because the main task often has time-consuming serial initialization code before spawning tasks. If the main task is executed by a slow core, it will increase the makespan of the program. To exclude the impact of this optimization in WATS, we make all other schedulers (Cilk, PFT, and RTS) launch the main task on the fastest core, though those schedulers may launch the main task on a randomly chosen core. If the chosen core is slow, which is very likely, their performance will be even worse.

Not surprisingly, WATS has one limitation. If most tasks in a parallel application execute the same function, the history-based task allocation algorithm will only find out a few task classes that cannot be evenly allocated to the c-groups. For example, divide-and-conquer programs such as *nqueens* are not suitable for WATS. To cope with this problem, we have modified the compiler *cilk2c* to check for the divide-and-conquer programs at compile time by analyzing the task generating pattern in the source code. If any function in the source code generates new tasks that run the same function as itself, the program is assumed to be a divide-and-conquer program. For divide-and-conquer programs, random task-stealing is used instead to schedule the program. Furthermore, if the program is also memory-bound, our previous CAB scheduler [16] can be adopted to improve its performance by reducing the cache misses. Therefore, the above limitation will not affect the applicability of WATS since the compiler can identify the class of programs that are suitable for WATS.

V. Related works

Researchers have shown the AMC architectures can achieve high performance and low power consumption [1], [2], [3], [4], [17]. An effective task scheduler is essential for parallel applications to make good use of the AMC architectures. However, the task scheduling policies, such as task-sharing and task-stealing adopted in current parallel programming environments, suffer from the problem of unbalanced workloads in AMC due to the assumption that all cores have equal performance. To our best knowledge, no previous study had addressed the scheduling problem in parallel programming environments where applications that are comprised of parallel tasks with different workloads can perform efficiently in AMC.

Many studies on scheduling in AMC focus on resource allocation at the OS level [18], [19], [20], [21], [22], [23], [24]. They aim to achieve high system throughput by balancing the hardware resources (e.g., cores, caches) among different programs. In [25], several phase co-scheduling policies are proposed for the OS to improve the overall throughput by reducing the conflicts among the phases of different threads. In [26], age-based scheduling is proposed to schedule the threads with larger remaining time to fast cores in AMC. [27] proposes a bias scheduling which matches threads to the right type of cores through dynamically monitoring the bias of the threads in order to maximize the system throughput. All of the above studies have not considered the scheduling problem in parallel applications that WATS has addressed in AMC.

Some recent studies addressed specific aspects of task scheduling of parallel applications in AMC. For example, in [28], ACS (Accelerated Critical Sections) is proposed to accelerate the execution of critical sections by migrating the threads with critical sections to fast cores. In [29], a speed balancing algorithm is proposed to manage the migration of threads so that each thread has a fair chance to run on the fastest core available. Instead of balancing the workloads, the algorithm balances the time of a thread executing on faster and slower cores. The downside of this work is that it assumes all threads have the same workload. Therefore, it cannot work for parallel tasks with different workloads as WATS does.

The only work that addresses the general scheduling problem in parallel applications is the random task-snatching [12] (i.e., RTS in Section IV-A), though it addresses the problem in the context of an Asymmetric Multi-Processor (AMP), which is similar to the context of AMC. RTS presents a model where each processor maintains an estimation of its speed. The model allows a fast core to snatch tasks randomly from a slow core when the fast core is idle and the task pool of the slow core is empty. As shown before, RTS cannot balance tasks as well as WATS due to its lack of workload information about the tasks.

Task-stealing has been extensively studied and adopted by parallel programming environments [5], [6], [7], [8], [15], [30], [31], though it does not perform well in AMC. An extension to task stealing for improving cache performance in multicore architectures has recently been proposed [16]. The preference-based task-stealing policy in WATS is a novel extension to task stealing to balance workloads among different groups of cores in AMC.

Task-stealing has also been extended to distributed systems. In [32], the authors extend task-stealing for large scale scientific applications on large distributed system (e.g. Blue Gene/P) in X10. In [33], high-level compiler optimizations and transformations are performed on the X10 programs to reduce communication and synchronization overheads of task stealing in distributed system. In [34], a *lifeline graph* that connects threads into k-dimensional hypercubes is proposed to provide high performance task stealing and active distributed termination in distributed memory architectures. However, these extensions to task-stealing are not relevant to the problem in AMC we address in this paper.

VI. Contributions and Conclusions

The contributions of this paper are as follows.

- We have identified, defined, and formalized the problem of unbalanced workloads in AMC architectures.
- We have analyzed the load-balancing problem and given theoretical guidance to optimal task allocation in AMC.
- We have proposed a history-based task allocation algorithm that can allocate tasks in AMC near-optimally.
- We have proposed a novel preference-based task-stealing policy that can effectively balance workloads among different groups of cores.
- Based on the above techniques, we have implemented a task scheduler, WATS, which achieves a performance gain of up to 64% compared to the random task stealing approach commonly employed.

AMC architectures are promising due to their high performance and power efficiency. It is essential for parallel applications to run on AMC architectures efficiently. Though task scheduling policies like task-stealing work efficiently for parallel applications in symmetric multicore architectures, they cannot balance the workloads well in AMC since they have no knowledge of task workloads and schedule tasks randomly to the performance-asymmetric cores.

From our theoretical analysis, we know that the initial optimal task allocation is more crucial to the makespan than any rescuing means for a non-optimal allocation and that static task allocation can produce near-optimal allocation if the workloads of the tasks are known. Therefore, we propose history-based task allocation that takes advantage of the static allocation by using the historical statistics of the tasks to predict the workloads and patterns of future tasks. From our experiments we showed that the history-based task allocation can produce near-optimal allocation and its extra overhead is small.

For any occasional inaccurate or incorrect allocation of tasks, the preference-based task-stealing policy comes to play. It can remedy any slightly unbalanced allocation and effectively schedule tasks among c-groups through preference-based stealing.

The experimental results show that our techniques adopted in WATS are effective and our approach to the scheduling problem in AMC is valid.

One potential avenue of future work is to explore near-optimal task scheduling in heterogeneous multi-core architectures that have heterogeneous accelerators (e.g., GPU or streaming processor). To schedule tasks in heterogeneous multi-core architectures, we can divide parallel tasks into task clusters according to their internal features and the hardware features. The task clusters will be allocated to the most suitable accelerators that can complete them in the shortest time. For example, we can schedule memory-bound tasks to cores with large and fast caches, but schedule data-parallel tasks to GPU or streaming processors. Another promising future research avenue is to investigate energy-aware task schedulers that would scale down the speed of the cores for memory-bound tasks with the assistance of DVFS. It would be interesting to find out how much energy will be saved and how much performance will be degraded, so that we can make the best tradeoff between energy and performance.

Acknowledgement

This work was partially supported by NFSC (Grant No. 60811130528, 61003012). We would like to thank David Eyers for his comments and suggestions. Quan Chen would like to thank the Department of Computer Science, University of Otago for funding and hosting his study.

References

[1] R. Kumar, D. Tullsen, N. Jouppi, and P. Ranganathan, "Heterogeneous chip multiprocessors," *Computer*, vol. 38, no. 11, pp. 32–38, 2005.

[2] 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 *Proceedings of the 31st annual International Symposium on Computer Architecture (ISCA'04)*. IEEE Computer Society, 2004.

[3] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, "The impact of performance asymmetry in emerging multicore architectures," in *Proceedings of the 32nd annual International Symposium on Computer Architecture (ISCA'05)*. IEEE Computer Society, 2005, pp. 506–517.

[4] M. Hill and M. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[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," *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, pp. 55–69, Aug. 1996.

[6] C. Leiserson, "The Cilk++ concurrency platform," in *Proceedings of the 46th Annual Design Automation Conference*. ACM, 2009, pp. 522–527.

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

[8] D. Lea, "A Java fork/join framework," in *Proceedings of the ACM 2000 conference on Java Grande*. ACM, 2000, pp. 36–43.

[9] J. Lee and J. Palsberg, "Featherweight X10: a core calculus for async-finish parallelism," in *Proceedings of the 15th ACM SIGPLAN symposium on Principles and Practice Of Parallel Programming (PPoPP'10)*. ACM, 2010, pp. 25–36.

[10] 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*, vol. 20, no. 3, pp. 404–418, 2009.

[11] R. Blumofe and C. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999.

[12] M. Bender and M. Rabin, "Scheduling Cilk multithreaded parallel programs on processors of different speeds," in *Proceedings of the 12nd annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'00)*. ACM, 2000, pp. 13–21.

[13] J. Adams, E. Balas, and D. Zawack, "The shifting bottleneck procedure for job shop scheduling," *Management science*, pp. 391–401, 1988.

[14] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI'98)*. Montreal, Canada: ACM, Jun. 1998, pp. 212–223.

[15] Y. Guo, R. Barik, R. Raman, and V. Sarkar, "Work-first and help-first scheduling policies for async-finish task parallelism," in *Proceedings of the 2009 IEEE International Parallel and Distributed Processing Symposium (IPDPS'09)*. IEEE Computer Society, 2009, pp. 1–12.

[16] Q. Chen, Z. Huang, M. Guo, and J. Zhou, "CAB: Cache-aware Bi-tier task-stealing in Multi-socket Multi-core architecture," in *40th International Conference on Parallel Processing (ICPP'11)*. Taipei, Taiwan: IEEE, 2011.

[17] R. Kumar, K. I. Farkas, N. P. Jouppi, P. Ranganathan, and D. M. Tullsen, "Single-isa heterogeneous multi-core architectures: The potential for processor power reduction," in *36th Annual International Symposium on Microarchitecture (MICRO'03)*. IEEE Computer Society, 2003.

[18] M. De Vuyst, R. Kumar, and D. Tullsen, "Exploiting unbalanced thread scheduling for energy and performance on a cmp of smt processors," in *Proceedings of the 2006 IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*. IEEE, 2006, pp. 10–20.

[19] A. Rosenberg and R. Chiang, "Toward understanding heterogeneity in computing," in *Proceedings of the 2010 IEEE International Parallel and Distributed Processing Symposium (IPDPS'10)*. IEEE, 2010, pp. 1–10.

[20] M. Bhadauria and S. McKee, "An approach to resource-aware co-scheduling for cmps," in *Proceedings of the 24th ACM International Conference on Supercomputing (ICS'10)*. ACM, 2010, pp. 189–199.

[21] T. Li, D. Baumberger, D. Koufaty, and S. Hahn, "Efficient operating system scheduling for performance-asymmetric multi-core architectures," in *Proceedings of the 2007 ACM/IEEE Conference on SuperComputing (SC'07)*. ACM, 2007, pp. 1–11.

[22] D. Shelepov, J. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. Huang, S. Blagodurov, and V. Kumar, "Hass: a scheduler for heterogeneous multicore systems," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 2, pp. 66–75, 2009.

[23] T. Morad, A. Kolodny, and U. Weiser, "Scheduling multiple multithreaded applications on asymmetric and symmetric chip multiprocessors," in *2010 Third International Symposium on Parallel Architectures, Algorithms and Programming (PAAP'10)*. IEEE, 2010, pp. 65–72.

[24] O. Khan and S. Kundu, "A self-adaptive scheduler for asymmetric multi-cores," in *Proceedings of the 20th symposium on Great Lakes Symposium on VLSI (GLSVLSI '10)*. ACM, 2010, pp. 397–400.

[25] A. El-Moursy, R. Garg, D. Albonesi, and S. Dwarkadas, "Compatible phase co-scheduling on a cmp of multi-threaded processors," in *Proceedings of the 2006 IEEE International Parallel and Distributed Processing Symposium (IPDPS'06)*. IEEE, 2006, pp. 10–pp.

[26] N. Lakshminarayana, J. Lee, and H. Kim, "Age based scheduling for asymmetric multiprocessors," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 2009, p. 25.

[27] D. Koufaty, D. Reddy, and S. Hahn, "Bias scheduling in heterogeneous multi-core architectures," in *Proceedings of the 5th European conference on Computer systems (EuroSys'10)*. ACM, 2010, pp. 125–138.

[28] M. Suleman, O. Mutlu, M. Qureshi, and Y. Patt, "Accelerating critical section execution with asymmetric multi-core architectures," in *Proceeding of the 14th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. ACM, 2009, pp. 253–264.

[29] S. Hofmeyr, C. Iancu, and F. Blagojević, "Load balancing on speed," in *Proceedings of the 15th ACM SIGPLAN symposium on Principles and Practice Of Parallel Programming (PPoPP'10)*. ACM, 2010, pp. 147–158.

[30] D. Leijen, W. Schulte, and S. Burckhardt, "The design of a task parallel library," *ACM SIGPLAN Notices*, vol. 44, no. 10, pp. 227–242, 2009.

[31] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "Slaw: a scalable locality-aware adaptive work–stealing scheduler," in *Proceedings of the 2010 IEEE International Parallel and Distributed Processing Symposium (IPDPS'10)*, 2010.

[32] J. Milthorpe, V. Ganesh, A. P. Rendell, and D. Grove, "X10 as a parallel language for scientific computation: practice and experience," in *Proceedings of the 2011 IEEE International Parallel and Distributed Processing Symposium (IPDPS'11)*. IEEE Computer Society, 2011.

[33] R. Barik, J. Zhao, D. Grove, I. Peshansky, Z. Budimlić, and V. Sarkar, "Communication Optimizations for Distributed-Memory X10 Programs," in *Proceedings of the 2011 IEEE International Parallel and Distributed Processing Symposium (IPDPS'11)*. IEEE Computer Society, 2011.

[34] V. Saraswat, P. Kambadur, S. Kodali, D. Grove, and S. Krishnamoorthy, "Lifeline-based global load balancing," in *Proceedings of the 16th ACM symposium on Principles and Practice Of Parallel Programming (PPoPP'11)*. ACM, 2011, pp. 201–212.