Adaptive Workload-Aware Task Scheduling for Single-ISA Asymmetric Multicore Architectures

QUAN CHEN and MINYI GUO, Department of Computer Science and Engineering,
Shanghai Jiao Tong University

Single-ISA Asymmetric Multicore (AMC) architectures have shown high performance as well as power efficiency. However, current parallel programming environments do not perform well on AMC because they are designed for symmetric multicore architectures in which all cores provide equal performance. Their random task scheduling policies 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 article proposes an adaptive Workload-Aware Task Scheduler (WATS) that consists of a history-based task allocator and a preference-based task scheduler. The history-based task allocator 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 scheduler, which schedules 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 allocator. Experimental results show that WATS can improve both the performance and energy efficiency of task-based applications, with the performance gain up to 66.1% compared with traditional task schedulers.

Categories and Subject Descriptors: D.4.1 [**Operating Systems**]: Process Management

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Task grouping, history-based task allocation, dynamic task scheduling

ACM Reference Format:
Quan Chen and Minyi Guo. 2014. Adaptive workload-aware task scheduling in single-ISA asymmetric multicore architectures. ACM Trans. Architec. Code Optim. 11, 1, Article 8 (February 2014), 25 pages.
DOI: http://dx.doi.org/10.1145/2579674

This article is extended from our previous conference paper, WATS: Workload-Aware Task Scheduling in Asymmetric Multicore Architecture [Chen et al. 2012a], which was published in IPDPS 2012. The 30% new material comes from the following aspects. (1) In the previous paper, WATS relied on a strong assumption that the percentage of tasks executing the same function among all tasks is almost the same during the execution of a parallel application. For batch-based programs, WATS in this article does not rely on this strong assumption; (2) We have updated WATS for supporting the generalized scenario without the aforementioned strong assumption. In addition, the new history-based task allocator does not rely on the frequencies of cores any more; thus, WATS can work in all single-ISA multicore architectures; (3) This article has also significantly enhanced the experimental evaluation. We have evaluated both the performance and the energy efficiency of WATS on much more asymmetric architectures.

Minyi Guo is the corresponding author of this article. Authors' addresses: Q. Chen and M. Guo, 3-119/3-417, SEIEE Building, No. 800 Dongchuan Road, Shanghai, China; email: chen-quan@situ.edu.cn; guo-my@cs.situ.edu.cn.

This work was partially supported by 863 program 2011AA01A202, Shanghai Excellent Academic Leaders Plan (No. 11XD1402900), Program for Changjiang Scholars and Innovative Research Team in University (IRT1158, PCSIRT) China, NSFC (Grant No. 60725208, 61003012).

DOI: http://dx.doi.org/10.1145/2579674

1. INTRODUCTION

Although chip manufacturers such as AMD and Intel keep producing new CPU chips with more symmetric cores, researchers are investigating alternative multicore organizations such as Asymmetric Multicore (AMC) architectures. In AMC architectures, individual cores have different computational capabilities [Kumar et al. 2004, 2005; Balakrishnan et al. 2005; Hill and Marty 2008].

AMC is attractive because it has the potential to improve system performance, to reduce power consumption, and to mitigate Amdahl's law [Kumar et al. 2005; Hill and Marty 2008]. Since an AMC architecture consists of a mix of fast cores and slow cores, it can better cater to applications with a heterogeneous mix of workloads [Kumar et al. 2004; Balakrishnan et al. 2005]. For example, fast, complex cores can be used to execute the serial code sections, whereas slow, simple cores can be used to crunch numbers in parallel, which is more power efficient. For example, Nintendo Wii and Nintendo DS use AMC processors. In addition, many modern multicore 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 multicore chip into a performance-asymmetric multicore chip.

Despite the rapid development of the AMC technology, current parallel programming environments, as listed later, still assume that 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 work-sharing or work-stealing policies for task scheduling. By dynamically scheduling the parallel tasks, the workloads can be balanced in multicore architectures. For example, Cilk [Blumofe et al. 1996], TBB [Reinders 2007], and X10 [Lee and Palsberg 2010] adopt work stealing, whereas OpenMP [Ayguadé et al. 2009] uses work sharing.

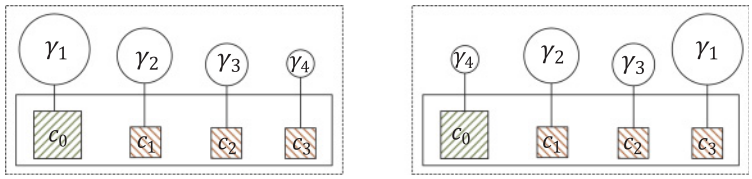
However, both work stealing and work sharing do not consider tasks' workloads when allocating tasks to different cores, which is not a problem for symmetric cores but can cause unbalanced workloads among asymmetric cores. For example, a long task may be scheduled to a slow core, whereas a short task is executed by a fast core. This problem of unbalanced workloads, which will be further discussed in detail in Section 2, can significantly 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 comprised of parallel tasks with different workloads can perform efficiently in AMC.

The techniques proposed in this article are used to improve the performance of parallel programs (especially batch-based programs and pipeline-based programs) on single-ISA AMC architectures where different types of cores in an AMC have the same Instruction Set Architecture (ISA). All of the tasks of a parallel program can be executed by any core in a single-ISA AMC architecture directly. Similar to Koufaty et al. [2010], Rosenberg and Chiang [2010], and Bhadauria and McKee [2010], we assume that different tasks have different requirements on the computation resources.

The rest of this article is organized as follows. Section 2 describes the problem of unbalanced workloads in AMC and the proposed solutions. Section 3 presents WATS, which comprises a history-based task allocator and a preference-based task scheduler. Section 4 evaluates WATS and provides limitations of WATS. Section 5 discusses related work. Section 6 summarizes our contributions and draws conclusions.

2. MOTIVATION AND SOLUTIONS

Let us use an example to explain the problem of unbalanced workloads in AMC. Suppose that a parallel application has four parallel tasks: γ_1, γ_2, γ_3, and γ_4. We assume



(a) Optimal allocation (b) Another possible allocation

Fig. 1. Two possible allocations of γ_1, γ_2, γ_3, and γ_4.

that the application runs on an AMC architecture as shown in Figure 1, with one fast core (c_0) and three slow cores (c_1, c_2, and c_3). Suppose that γ_1, γ_2, γ_3, and γ_4 take times f_1, f_2, f_3, and f_4 on the fast core c_0, respectively, and that $f_1 > f_2 > f_3 > f_4$. As the counterpart, γ_1, γ_2, γ_3, and γ_4 take times s_1, s_2, s_3, and s_4 on the slow cores, respectively. We can reasonably deduce that $s_1 > f_1, s_2 > f_2, s_3 > f_3$, and $s_4 > f_4$. Without loss of generality, we further assume that $f_1 > s_2$, $f_1 > s_3$, and $f_1 > s_4$.

Figure 1 shows two possible allocations of γ_1, γ_2, γ_3, and γ_4 to the cores. Figure 1(a) is an optimal allocation where γ_1 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 γ_1, γ_2, γ_3, and γ_4 is $T_{opt} = max\{f_1, s_2, s_3, s_4\} = f_1$. Because $f_1 < s_1$, we can find that $T_{opt} < s_1$.

However, with traditional task scheduling policies such as work stealing, γ_1, γ_2, γ_3, and γ_4 are likely to be allocated as in Figure 1(b), where γ_3 is allocated to the fast core but the long task γ_1 is scheduled to a slow core. In this case, the makespan for γ_1, γ_2, γ_3, and γ_4 is $T_{bad} = max\{s_1, s_2, f_3, s_4\} \geq s_1 > f_1 = T_{opt}$. Obviously, allocating a long task to a slow core can degrade the overall performance seriously.

Some studies (e.g., Bender and Rabin [2000]) 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 Figure 1(b), c_0 is allowed to snatch γ_1 from c_3 after finishing γ_4. Suppose that c_0 snatches γ_1 from c_3 after finishing γ_4 (which takes time f_4). c_0 still needs $(\frac{s_1-f_4}{s_1}) \times f_1$ to finish γ_1 because c_3 has only finished $\frac{s_1-f_4}{s_1}$ of γ_1. Let Δ_s represent the time of the snatching operation. Then, the overall time for c_0 to finish both γ_4 and γ_1 is $f_4 + \frac{s_1-f_4}{s_1} \times f_1 + \Delta_s$. Therefore, with the snatching policy, the makespan for γ_1, γ_2, γ_3, and γ_4 is $T_{res} = max\{f_4 + \frac{s_1-f_4}{s_1} \times f_1 + \Delta_s, s_2, s_3, f_4\}$. Because $f_4 + \frac{s_1-f_4}{s_1} \times f_1 + \Delta_s - f_1 > \frac{f_1}{s_1} \times f_4 + \frac{s_1-f_4}{s_1} \times f_1 - f_1 + \Delta_s = \frac{f_4}{s_1} \times f_1 + \frac{s_1-f_4}{s_1} \times f_1 - f_1 + \Delta_s = \Delta_s$, we can deduce that $f_4 + \frac{s_1-f_4}{s_1} \times f_1 + \Delta_s > f_1$. In addition, since f_1 is larger than s_2, s_3, and f_4, $T_{res} = f_4 + \frac{s_1-f_4}{s_1} \times f_1 + \Delta_s > f_1 = T_{opt}$ and the rescuing policy is still not as efficient as the optimal allocation.

Furthermore, since $T_{res} - T_{bad} = f_4 + \frac{s_1-f_4}{s_1} \times f_1 + \Delta_s - s_1 = (s_1 - f_4) \times (\frac{f_1}{s_1} - 1) + \Delta_s$ and $(s_1 - f_4) \times (\frac{f_1}{s_1} - 1) < 0$, if the system knows the execution time of each task on all the cores and Δ_s is not too large, the snatching policy can improve the performance of random scheduling.

However, the execution time of the tasks on different cores are unknown to the existing random schedulers. Therefore, 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 Figure 1(b), with the random snatching, the worst case could be that c_0 first snatches γ_2 and γ_3 before snatching γ_1, where the makespan is larger.

In summary, the knowledge of tasks' execution time on different cores is essential to optimal task scheduling in AMC. This knowledge can help a scheduler allocate long

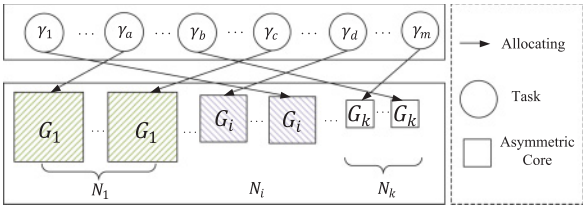


Fig. 2. The optimal task allocation problem in AMC. Allocate m independent tasks with different workloads to k c-groups with different computational capacities.

Table I. ETT of the Program That Has m Tasks on an AMC with k C-groups

Tasks \ C-groups	G_1	G_2	...	G_i	...	G_k
γ_1	t_{11}	t_{12}	...	t_{1i}	...	t_{1k}
γ_2	t_{21}	t_{22}	...	t_{2i}	...	t_{2k}
...	...	...	...	...	...	...
γ_j	t_{j1}	t_{j2}	...	t_{ji}	...	t_{jk}
...	...	...	...	...	...	...
γ_m	t_{m1}	t_{m2}	...	t_{mi}	...	t_{mk}

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 nonoptimal allocation.*

We generalize the task allocation problem, assuming that the execution time of tasks on all cores are known. We will give theoretical analysis on the optimal task allocation, which will guide our design and implementation of task scheduling in AMC.

Figure 2 illustrates the general problem of optimal task allocation in AMC. Suppose that there are m independent tasks $(\gamma_1, \ldots, \gamma_m)$ with different workloads and an AMC with k types of cores. We group cores of the same type into a core group (denoted as c-group). We use $G_1, \ldots, G_k$ to represent the k c-group in descending order of their computational capacities and use N_i $(1 \leq i \leq k)$ to represent the number of cores in G_i. The problem can be expressed as *how to divide the m tasks into k groups that are assigned to the k c-groups, respectively, so that the makespan is minimum?* Once tasks are assigned to c-groups, many existing task scheduling policies (e.g., work sharing and work stealing) can be adopted to balance workloads among symmetric cores in the same c-group.

To solve the problem, we construct an Execution Time Table (ETT) for the parallel program that has m tasks on the AMC that has k c-groups in Table I. The item t_{ji} in row γ_j and column G_i is the expected execution time of γ_j on a core in c-group G_i.

2.1. Theoretical Ideal Solution

The following theorem provides theoretical guidance to optimal task allocation.

THEOREM 2.1. *For tasks $\gamma_1, \ldots, \gamma_m$, if $\gamma_{p_{i-1}+1}, \ldots, \gamma_{p_i}$ $(1 \leq i \leq k, p_0 = 0, p_k = m)$ are allocated to c-group G_i, their makespan is minimum only when $p_1, \ldots, p_{k-1}$ satisfy*

$$\sum_{n=1}^{p_1} t_{n1} : \cdots : \sum_{n=p_{i-1}+1}^{p_i} t_{ni} : \cdots : \sum_{n=p_{k-1}+1}^{m} t_{nk} = N_1 : \cdots : N_i : \cdots : N_k \tag{1}$$

Moreover, the task allocation is optimal and the optimal makespan $T_{opt} = \frac{\sum_{n=1}^{p_1} t_{n1}}{N_1} = \cdots = \frac{\sum_{n=p_{i-1}+1}^{p_i} t_{ni}}{N_i} \cdots = \frac{\sum_{n=p_{k-1}+1}^{m} t_{nk}}{N_k}$.

Table II. Allocate m Tasks with Different Workloads to k C-groups

C-groups / Tasks	G_1	G_2	...	G_i	...	G_k
γ_1	t_{11}	t_{12}	...	t_{1i}	...	t_{1k}
...	...	...	...	...	...	...
γ_{p_1}	$t_{(p_1)1}$	$t_{(p_1)2}$	...	$t_{(p_1)i}$	...	$t_{(p_1)k}$
γ_{p_1+1}	$t_{(p_1+1)1}$	$t_{(p_1+1)2}$	...	$t_{(p_1+1)i}$	...	$t_{(p_1+1)k}$
...	...	...	...	...	...	...
γ_{p_2}	$t_{(p_2)1}$	$t_{(p_2)2}$	...	$t_{(p_2)i}$	...	$t_{(p_2)k}$
...	...	...	...	...	...	...
$\gamma_{p_{i-1}+1}$	$t_{(p_{i-1}+1)1}$	$t_{(p_{i-1}+1)2}$	...	$t_{(p_{i-1}+1)i}$	...	$t_{(p_{i-1}+1)k}$
...	...	...	...	...	...	...
γ_{p_i}	$t_{(p_i)1}$	$t_{(p_i)2}$	...	$t_{(p_i)i}$	...	$t_{(p_i)k}$
...	...	...	...	...	...	...
$\gamma_{p_{k-1}+1}$	$t_{(p_{k-1}+1)1}$	$t_{(p_{k-1}+1)2}$	...	$t_{(p_{k-1}+1)i}$	...	$t_{(p_{k-1}+1)k}$
...	...	...	...	...	...	...
γ_m	t_{m1}	t_{m2}	...	t_{mi}	...	t_{mk}

PROOF. Straightforward. If tasks are divided into groups in Equation 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_{opt} 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 $\frac{\sum_{n=1}^{p_1} t_{n1}}{N_1} = \cdots = \frac{\sum_{n=p_{i-1}+1}^{p_i} t_{ni}}{N_i} \cdots = \frac{\sum_{n=p_{k-1}+1}^{m} t_{nk}}{N_k} = T_{opt}$. □

2.2. Proposed Solution

It is not feasible to find the ideal solution to Theorem 2.1 because one may not exist in real situations. Even if one does exist, the problem is defined as *the minimum maksspan problem on uniform parallel machines* [Liu and Liu 1974] which is NP-hard.

Due to the reasons presented earlier, we relax the conditions of Theorem 2.1 and propose a heuristical solution for the task allocation problem in AMC, as shown in Table II.

In the solution, the m independent tasks are sorted in descending order of their execution time on the fastest core (any core in G_1). If the fastest core needs longer time to execute a task a than another task b, the workload of a is heavier than the workload of b. Based on the sorted tasks, we choose $p_1, \ldots, p_{k-1}$ to divide the m tasks into k groups that are allocated to the k c-groups (i.e., $G_1, \ldots, G_k$) according to Algorithm 1.

We assume that there are enough tasks to be allocated to the c-groups and that each c-group will be allocated at least 1 task (i.e., $p_i < p_k$ if $i < k$). Observed from Table II, once $p_1, \ldots, p_{k-1}$ are determined, the m tasks are divided into k groups. Because the values of $p_1, \ldots, p_{k-1}$ could be $1, 2, \ldots, m-1$, the number of possible choices of dividing the m tasks equals the number of possible choices of selecting $k-1$ numbers from $1, 2, \ldots, m-1$. The earlier problem is a combination problem, and the overall number of choices is C_{m-1}^{k-1}. Therefore, in Algorithm 1, we compare the estimated makespan of the tasks for each of all C_{m-1}^{k-1} combinations of $p_1, \ldots, p_{k-1}$ and choose the combination of $p_1, \ldots, p_{k-1}$ that results in the minimum makespan.

C_{m-1}^{k-1} will be very large if both m and k are large. In the worst case, Algorithm 1 is of an exponential time complexity. Fortunately, AMC architectures only have two types

of cores in most cases (i.e., $k = 2$) [Van Craeynest et al. 2012; Joao et al. 2012; Saez et al. 2010b]. If $k = 2$, $C_{m-1}^{k-1} = C_{m-1}^{1} = m - 1$, which increases with the number of tasks linearly. In addition, by grouping tasks into task classes, WATS can further greatly reduce m and thus can further significantly reduce C_{m-1}^{k-1} (the details will be explained in Section 3.2.3). Because both m and k are small in WATS, the overhead of Algorithm 1 is negligible for real-world applications and real AMC architectures.

ALGORITHM 1: Static near-optimal task allocation

Input: A set of tasks $\{\gamma_1, \ldots, \gamma_m\}$; The ETT of tasks on k c-groups: t[m][k]
Input: The numbers of cores in c-groups $G_1, \ldots, G_k$: $N_1, \ldots, N_k$
Output: p[k-1]: $\{p_1, \ldots, p_{k-1}\}$

Func.: `AllocateTask`
int p[k-1], q[k-1] ; `// q[k-1] stores the to-be-evaluated combination of` $p_1, \ldots, p_{k-1}$
int i= 0, min_span = MAXMUM_INT;
while *Not all the settings are evaluated* **do**
  Get a new setting from C_{m-1}^{k-1} possible settings and update q[k-1];
  if *any of* $\{\frac{\sum_{n=0}^{q[0]} t[n][0]}{N_1}, \ldots, \frac{\sum_{n=q[k-2]+1}^{m-1} t[n][k-1]}{N_k}\} > min\_span$ **then** continue;
  else min_span = max $\{\frac{\sum_{n=0}^{q[0]} t[n][0]}{N_1}, \ldots, \frac{\sum_{n=q[k-2]+1}^{m-1} t[n][k-1]}{N_k}\}$; Copy q[k-1] to p[k-1];
return p[k-1];

In the heuristical near-optimal solution presented, we assume that the ETT of the program has been constructed and that all of the items in ETT are known. However, in real parallel applications, this assumption is not valid because these information is not known until they complete. How to apply the theoretical solution to parallel programming environments is a challenging issue.

In WATS, we propose a *history-based task allocator* to initially allocate tasks to the right c-groups. Tasks are classified into *task classes* according to their function names. Instead of allocating tasks directly, we allocate the task classes to different c-groups. For the same function f, we can collect the average execution time of the f-named tasks on cores in every c-group, respectively, in the history. Because the average execution time of each task class in every c-group is known from history, we can adopt Algorithm 1 to allocate the functions to different c-groups. Based on this allocation, tasks will be allocated to the c-group where its function name is allocated.

In seldom cases where history cannot precisely predict the future, the allocation suggested by the earlier history-based task allocator is only an approximation of the optimal allocation. In order to further balance the workload, we propose a *preference-based task scheduler* to adjust the workloads dynamically among different c-groups.

If the number and workloads of tasks in the same task class are totally repeatable and can be estimated accurately, similar to our history-based task allocator, some other task allocating algorithms [Hochbaum and Shmoys 1988; Miguet and Pierson 1997] can provide a near optimal scheduling. However, for real applications, the workloads of tasks in the same task class are similar but are not totally repeatable. As far as we know, the linear programming–based technique cannot tolerate the nonrepeatability due to its static scheduling. On the contrary, WATS can tolerate some nonrepeatability due to the dynamic preference-based task scheduler.

3. WORKLOAD-AWARE TASK SCHEDULING

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 allocator and the preference-based task scheduler are used to fulfill the philosophy.

Without loss of generality, we assume that the asymmetric cores in AMC can be divided into k c-groups $G_1, \ldots, G_k$, where G_i has N_i cores, and the cores in G_i are faster than the cores in G_j if $i < j$.

3.1. Overview of WATS

As presented earlier, in WATS, instead of allocating the dynamically spawned tasks, we allocate the task classes to different c-groups. To support the strategy, WATS creates one task pool for each task class to store its tasks. When a task γ with a function name f is generated, its task pool is checked first. If the task pool for f-named tasks exists, γ is pushed into the correspondence task pool. If there is no task pool for f-named tasks, then a new task pool is created and γ is pushed into the new task pool.

Generally, we use a data structure $TC(f, ipc, n_1, \ldots, n_k, t_1, \ldots, t_k)$ to represent a task class, where f is the function name, ipc is the Instruction Per Cycle (IPC) of a task in the task class on a core in the fastest c-group G_1, n_i $(1 \leq i \leq k)$ is the number of tasks executed by cores in c-group G_i in history, and t_i $(1 \leq i \leq k)$ is the estimated execution time of a task in the task class on a core in c-group G_i. Note that in any task class $TC(f, ipc, n_1, \ldots, n_k, t_1, \ldots, t_k)$, for any $1 \leq i \leq k$, n_i and t_i cannot be obtained directly because the tasks are spawned dynamically and their real execution time on different cores cannot be obtained until they are completed.

To allocate task classes to c-groups appropriately, the key issue is to obtain ipc, n_i $(1 \leq i \leq k)$, and t_i $(1 \leq i \leq k)$ of all task classes precisely. Targeting this issue, WATS uses a history-based task allocator to collect ipc, n_i $(1 \leq i \leq k)$, and t_i $(1 \leq i \leq k)$ of all task classes based on historical statistics. Once the information of all task classes are determined, the history-based task allocator can allocate the task classes to different c-groups near optimally using Algorithm 1.

After the task classes are allocated, WATS uses a preference-based task scheduler to balance tasks among cores in the same c-group and among different c-groups dynamically. In the preference-based task scheduler, once cores in one c-group finish all of the tasks allocated to the c-group, the cores help other c-groups to execute their tasks. To achieve this purpose, each core is given a *preference list* of task clusters (to be defined shortly). An idle core obtains a task according to the order of its preference list.

3.2. History-Based Task Allocator

During the execution of a parallel program, WATS assumes that tasks executing the same function in the current run have similar workloads. As for the assumption, although a function may show divergent behaviors depending on the inputs, the inputs of the tasks in the same task class in one run are often similar due to data parallelism. Empirically, in order to parallelize a serial program, the whole dataset of the serial program has often been divided into many equal-size data blocks, and each task will work on a single data block. Therefore, most well-designed data parallel programs obey this assumption. For example, in pipeline programs (such as Dedup and Ferret in Parsec benchmark suite [Bienia et al. 2008]), tasks in different stages run in parallel. Tasks in the same stage execute the same function and have similar workloads, but tasks in different stages execute different functions and have different workloads. For programs that do not obey this assumption (such as divide-and-conquer programs), traditional work-stealing strategy is used to schedule the programs.

Based on the assumption presented, WATS uses the historical statistics collected during the execution of a program to estimate the workloads of future tasks in the same run. WATS collects the execution time and IPC of all completed tasks.

3.2.1. Build Task Classes for Nonbatch Programs. In parallel programs whose tasks are not processed in batches, such as pipeline programs, the parallel tasks are generated dynamically at runtime. Because the tasks are spawned continually, it is not possible to group the unprocessed tasks and determine the appropriate allocation directly.

Therefore, for nonbatch programs, WATS further assumes that the percentage of tasks executing the same function among all tasks is almost the same during the execution of a parallel application. As for this assumption, in many signal-processing programs, different signals are input into the programs at a constant rate, where tasks processing different signals are created at a constant rate. As another example, most pipeline programs also obey this assumption. In pipeline programs, the data are divided into many data chunks and the data processing are divided into several stages. Because a task is launched for a data chunk at every stage and every data chunk needs to go through all of the stages, the assumption is approved.

Under this assumption, the near-optimal task allocation for the completed tasks are also near optimal for the future tasks. Therefore, in this case, the history-based core allocator searches the near-optimal task allocation for the completed tasks instead and then allocates the newly spawned tasks in the same allocation strategy. To find the appropriate allocation for the completed tasks, the tasks completed in history are also organized as task classes according to their function names.

We still use $TC(f, ipc, n_1, \ldots, n_k, t_1, \ldots, t_k)$ to represent a task class that is comprised of the completed tasks. In WATS, the task classes of completed tasks are updated in a timely manner. Once a task γ is completed, WATS collects its execution time and its IPC. Suppose that the execution time and IPC of γ are t_γ and ipc_γ, respectively. If γ is executed by a core in G_1, then its task class $TC(f, ipc, n_1, \ldots, n_k, t_1, \ldots, t_k)$ is updated to $TC(f, \frac{ipc \times n_1 + ipc_\gamma}{n_1+1}, n_1+1, \ldots, n_k, \frac{t_1 \times n_1 + t_\gamma}{n_1+1}, \ldots, t_k)$. If γ is executed by a core in G_i $(i > 1)$, n_i in its task class is updated to $n_i + 1$, and t_i in its task class is updated to $\frac{t_i \times n_i + t_1 \times \frac{ipc}{ipc_\gamma}}{n_i+1}$. If there is no such class, a new task class $TC(f, ipc, n_1, \ldots, n_k, t_1, \ldots, t_k)$ is created for f. In the newly created task class, $n_i = 1, t_i = t_\gamma$ and all other items are 0.

Note that for task γ executed by a core in c-group G_i, WATS does not use its real execution time t_γ to update t_i in its task class $TC(f, ipc, n, t_1, \ldots, t_i, \ldots, t_k)$ but uses its IPC ipc_γ to update t_i. This is because the execution time of a task on a core may increase due to events that the runtime system cannot control. This could be the case of the execution of an interrupt handler on the core where the task was scheduled, page-fault processing, or any bottom-half processing done by the operating system (OS) (load balancing or I/O-related operations). Fortunately, we can use $t_1 \times \frac{ipc}{ipc_\gamma}$ to calculate the execution time of γ when it is not interrupted by any other events. Using the expected execution time of γ, the information in the task classes is accurate enough for construction of the ETT of the correspondence program.

Based on the information about the task classes, the next step is to allocate the task classes of the completed tasks to the k c-groups in Section 3.2.3.

3.2.2. Build Task Classes for Batch Programs. In batch programs, the parallel tasks are launched and processed in batches. Only when all tasks in a batch are completed, the program launches another batch of tasks.

If a batch program enters a new batch, in WATS, the workers do not execute the tasks immediately but let the program generate all tasks in the batch first. Because the tasks are divided into task classes and are distributed to different task pools when they are generated, once all of the tasks in the batch are spawned, WATS gets many task classes of unexecuted tasks and the correspondence task pools.

WATS uses a simplified data structure $TC(f, ipc, n, t_1, \ldots, t_k)$ to represent a task class in a batch program, in which n is the number of f-named tasks in the current batch and t_i $(1 \le i \le k)$ is the estimated execution time of the tasks in the task class on a core

Table III. ETT of a Nonbatch Program with m Task Classes on an AMC with k C-groups

C-groups / Task classes	G_1	...	G_a	...	G_k
TC_1	$\sum_{j=1}^{k} n_{1j} \cdot t_{11}$	...	$\sum_{j=1}^{k} n_{1j} \cdot t_{1a}$	...	$\sum_{j=1}^{k} n_{1j} \cdot t_{1k}$
TC_2	$\sum_{j=1}^{k} n_{2j} \cdot t_{21}$	...	$\sum_{j=1}^{k} n_{2j} \cdot t_{2a}$	...	$\sum_{j=1}^{k} n_{2j} \cdot t_{2k}$
...	...	...	...	...	...
TC_i	$\sum_{j=1}^{k} n_{ij} \cdot t_{i1}$	...	$\sum_{j=1}^{k} n_{ij} \cdot t_{ia}$	...	$\sum_{j=1}^{k} n_{ij} \cdot t_{ik}$
...	...	...	...	...	...
TC_m	$\sum_{j=1}^{k} n_{mj} \cdot t_{m1}$	...	$\sum_{j=1}^{k} n_{mj} \cdot t_{ma}$	...	$\sum_{j=1}^{k} n_{mj} \cdot t_{mk}$

Table IV. ETT of a Batch Program with m Task Classes on an AMC with k C-groups

C-groups / Task classes	G_1	...	G_a	...	G_k
TC_1	$n_1 \cdot t_{11}$	...	$n_1 \cdot t_{1a}$	...	$n_1 \cdot t_{1k}$
TC_2	$n_2 \cdot t_{21}$	...	$n_2 \cdot t_{2a}$	...	$n_2 \cdot t_{2k}$
...	...	...	...	...	...
TC_i	$n_i \cdot t_{i1}$	...	$n_i \cdot t_{ia}$	...	$n_i \cdot t_{ik}$
...	...	...	...	...	...
TC_m	$n_m \cdot t_{m1}$	...	$n_m \cdot t_{ma}$	...	$n_m \cdot t_{mk}$

in c-group G_i. In $TC(f, ipc, n, t_1, \ldots, t_k)$, n can be collected by counting the number of tasks in the correspondence task pool. Based on the historical statistics, we calculate t_j $(1 \leq j \leq k)$ in $TC(f, ipc, n, t_1, \ldots, t_k)$. Let r represent the number of f-named tasks completed by cores in c-group G_i in history, and let $ipc_{j1}, \ldots, ipc_{jr}$ represent their IPCs. We can calculate t_j in $TC(f, ipc, n, t_1, \ldots, t_j, \ldots, t_k)$ in Equation 2:

$$t_j = t_1 \times \frac{ipc}{\sum_{m=1}^{r} ipc_{jm}/r}. \tag{2}$$

Once WATS collects n and calculates $t_1, \ldots, t_k$ for each task class in the batch, the history-based task allocator can allocate the task classes to the c-groups as follows.

3.2.3. Allocate Task Classes to c-Groups. Suppose that there are overall m task classes. If the parallel program is not a batch program, the m task classes are denoted by $TC_1(f_1, ipc_1, n_{11} \ldots, n_{1k}, t_{11}, \ldots, t_{1k}), \ldots, TC_m(f_m, ipc_m, n_{m1} \ldots, n_{mk}, t_{m1}, \ldots, t_{mk})$. Otherwise, if the parallel program is a batch program, the m task classes are denoted by $TC_1(f_1, ipc_1, n_1, t_{11}, \ldots, t_{1k}), \ldots, TC_m(f_m, ipc_m, n_m, t_{m1}, \ldots, t_{mk})$. Note that the m task classes $TC_1, \ldots, TC_m$ are sorted in the descending order of t_{i1} $(1 \leq i \leq m)$.

If the task classes are ready, we apply Algorithm 1 to divide the task classes into k groups and allocate them to the k c-groups accordingly. In order to apply Algorithm 1, we need to build ETT for the m task classes first. Table III and Table IV give the ETT for a nonbatch program and a batch program, respectively.

Similar to Table I, in Tables III and IV, the very item at row TC_i and column G_a represents the time needed by a core in c-group G_a to execute all tasks in task class TC_i. Recall that the expected execution time of a task in TC_i on a core in c-group G_a is t_{ia}. Based on the expected execution time, we can calculate the items in Table III and Table IV. For a nonbatch program, because $\sum_{j=1}^{k} n_{ij}$ tasks in task class TC_i are completed in history, the very item in row TC_i and column G_a is $\sum_{j=1}^{k} n_{ij} \cdot t_{ia}$. Meanwhile, for a batch program, because there are overall n_i tasks in task class TC_i are generated in the current batch, the very item in row TC_i and column G_a of Table IV is $n_i \cdot t_{ia}$.

Based on Tables III and IV, we apply 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

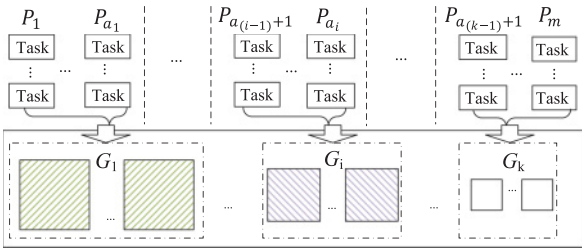


Fig. 3. Allocating m task classes to k c-groups in the history-based task allocator.

classes *task clusters*. Since task clusters and c-groups are a one-to-one mapping, for the sake of convenience, we use G_i to represent both a task cluster and a c-group in the following discussion. Figure 3 illustrates how the history-based task allocator works. In the figure, task pool P_j $(1 \le j \le m)$ stores tasks in task class TC_j.

As mentioned earlier, in Tables III and IV, the m task classes are sorted in the descending order of t_{i1} $(1 \le i \le m)$, not $\sum_{j=1}^{k} n_{ij} \cdot t_{i1}$ $(1 \le i \le m)$ or $n_i \cdot t_{i1}$ $(1 \le i \le m)$. In this way, if we unfold the task classes into tasks, the tasks in the new ETT are sorted in the descending order of their execution time on the fastest core as in Table I, which is the basis of Algorithm 1. However, if we unfold the task classes in Tables III and IV into tasks, the new ETTs will have $M = \sum_{a=1}^{m} \sum_{b=1}^{k} n_{ab}$ rows and $M = \sum_{a=1}^{m} n_a$ rows, respectively. In this case, Algorithm 1 has to check C_{M-1}^{k-1} possible combinations of $p_1, \ldots, p_{k-1}$ to search for the optimal allocation of tasks to c-groups. On the other hand, if the tasks are grouped into task classes as in Tables III and IV, Algorithm 1 only needs to check C_{m-1}^{k-1} possible combinations. Because a parallel program often has a great amount of tasks but only a small number of task classes, $m \ll M$ and $C_{m-1}^{k-1} \ll C_{M-1}^{k-1}$. Essentially, by grouping tasks into task classes, we make sure that the tasks executing the same function are allocated to the same c-group. In this way, we can greatly reduce the tries needed to search for the appropriate allocation of tasks to c-groups. For instance, suppose 200 tasks that can be classified into 10 task classes are completed. If the program runs on an AMC with four types of cores (normally, there are only two types of cores in an AMC), the number of combinations of $p_1, \ldots, p_{k-1}$ is reduced from $C_{199}^{3} = 1{,}293{,}699$ to $C_{9}^{3} = 84$ by grouping tasks into task classes. Therefore, the overhead of Algorithm 1 in WATS is small.

It is worth noting that all of the information used in the history-based task allocator is collected automatically. The number of cores in every c-group can be acquired from the OS. The execution time and IPC of a task are acquired at runtime. Once a task is completed, the information about its task class is updated as presented in Section 3.2.1.

In addition, the execution time of a memory-intensive task can vary from run to run due to the contention on shared resources. The contention may result in a slightly unbalanced workload in WATS. For a newly generated task, since we use the IPCs of all tasks completed in history in its task class to estimate its execution time, the calculated execution time is close to its real execution time. Even though the contention on shared resources incurs slight load unbalancing, the preference-based task scheduler can further balance the workload dynamically by scheduling the tasks at runtime.

3.3. Preference-Based Task Scheduler

WATS uses a preference-based task scheduler to balance workloads dynamically. In our situation, task scheduling is complex since there are overall m task pools, labeled as $P_1, \ldots, P_m$, corresponding to the m task classes $TC_1, \ldots, TC_m$.

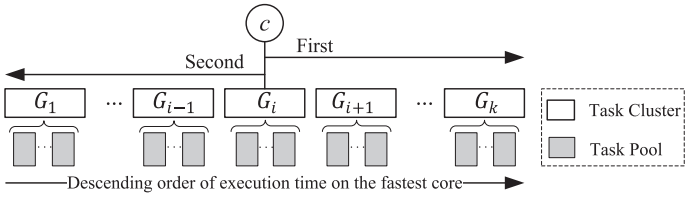


Fig. 4. Preference list of the cores in the c-group G_i.

We use a core c from the c-group G_i in Figure 3 as an example to explain the details of the preference-based task scheduler.

3.3.1. Scheduling within a c-Group. If c is free, it first tries to obtain tasks from the task pools that store tasks allocated to its c-group G_i (i.e., $P_{a_{(i-1)}+1}, \ldots, P_{a_i}$ as shown in Figure 3). Since multiple task pools are allocated to G_i, c needs to decide to obtain a task from which pool first.

A basic strategy for choosing a victim task pool to obtain a task is choosing task pools in the order of $P_{a_{(i-1)}+1}, \ldots, P_{a_i}$, which is the order of task classes in Figure 3. Only when the task pool P_j $(a_{(i-1)} + 1 \leq j < a_i)$ is empty does c try to obtain a task from the next task pool P_{j+1} until it gets a task.

This basic strategy is similar to a work-sharing strategy in, which all of the cores share a single task pool. In the basic strategy, cores in G_i try to execute all of the tasks in one task pool before moving to the next task pool allocated to G_i. Therefore, the basic strategy often causes serious lock contention on the task pools similar to work sharing, since many cores in G_i may try to lock the same task pool for obtaining new tasks. The serious lock contention may degrade the performance of WATS.

To reduce lock contention, we decide to use a strategy borrowed from random work-stealing that has been proved to be effective.[1] In WATS, if c is free, it randomly chooses a task pool P_j from $P_{a_{(i-1)}+1}, \ldots, P_{a_i}$ and tries to obtain a task from P_j. If P_j is empty, it randomly chooses another task pool and tries to obtain a task from the new chosen task pool until c gets a task. In this case, since there are multiple task pools for obtaining tasks, the lock contention is much lower and the performance would be better.

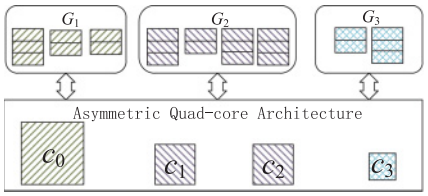
3.3.2. Scheduling among c-Groups. If all of the task pools allocated to G_i are empty, which means that all tasks allocated to G_i are completed, WATS allows c to execute tasks allocated to other c-groups in order to balance the workloads among different c-groups dynamically. The complexity arises when deciding which c-group to choose in this situation. The following preference-based strategy gives our solution.

In the preference-based task scheduler, each core is given a *preference list* of task clusters. Each task cluster contains multiple task pools. The preference list of a core contains all of the k task clusters that are ordered as detailed next.

For core c in the c-group G_i, its preference list is created as $\{G_i, G_{i+1}, \ldots, G_k, G_{i-1}, G_{i-2}, \ldots, G_1\}$ as shown in Figure 4.

The preference list in Figure 4 is generated based on the *help-the-weaker-first* principle. This principle can help reduce the makespan. For example, if a core obtains a task that is allocated to faster cores, it needs a long time to execute the obtained task, which may prolong the makespan. On the contrary, if a core obtains a task that is allocated to slower cores, it can execute the obtained task in a shorter time and relieve the pressure on slow cores. However, this preference list does not prevent slow cores to obtain tasks

[1]In random work stealing, each core has a task pool. When a core is free, a core randomly chooses a victim core and tries to steal a task from the victim core's task pool. In our scenario, there are multiple task pools as well, but they are associated with task classes and not cores.



(a) An example structure of WATS

C-group	Core	Preference list
G_1	c_0	$\{G_1, G_2, G_3\}$
G_2	c_1 & c_2	$\{G_2, G_3, G_1\}$
G_3	c_3	$\{G_3, G_2, G_1\}$

(b) Preference lists of the asymmetric cores

Fig. 5. An example runtime structure of WATS and the corresponding preference lists.

from fast cores. When the slow cores have no tasks, they can obtain tasks from the busy fast cores.

Once c decides to help c-group G_j ($1 \leq j \leq k$), it randomly selects the victim task pool from the task pools allocated to G_j following the same strategy described in Section 3.3.1. Algorithm 2 shows the preference-based task scheduling algorithm adopted by each core for obtaining a new task.

ALGORITHM 2: Preference-based task scheduling

Input: A core c from the c-group G_i
Input: c's preference list $\{G_i, \ldots, G_k, G_{i-1} \ldots, G_1\}$

Func.: `ObtainNewTask`
while *c has not obtained a task* **do**
  for *each* $G_j \in \{G_i, \ldots, G_k, G_{i-1} \ldots, G_1\}$ **do**
    while *not all the task pools allocated to* G_j *are empty* **do**
      c randomly chooses a task pool P_a allocated to G_j;
      c tries to obtain a task t from P_a;
      if *succeed* **then return** t;

Figure 5(a) shows an example runtime structure of WATS on an asymmetric quad-core architecture with three different types of cores. That is, there are three c-groups: G_1 (with core c_0), G_2 (with c_1 and c_2), and G_3 (with c_3).

Therefore, task classes are classified into three task clusters (G_1, G_2, and G_3) accordingly. The preference lists of the four cores are generated as in Figure 5(b), based on the *help-the-weaker-first* principle in Figure 4. For example, c_3 will always look for tasks from the G_3 pools first, which have the tasks that are allocated to c_3's c-group using the history-based task allocator. Then, it will search the G_2 pools and finally the G_1 pools.

3.4. Implementation

WATS has been implemented by modifying MIT Cilk, which consists of a compiler and a scheduler. MIT Cilk is one of the earliest open source parallel programming environments that implement work stealing [Frigo et al. 1998]. The Cilk compiler, named *cilk2c*, is a source-to-source translator that transforms a Cilk source program into a C program.

We have ported MIT Cilk to support the preference-based task scheduler. 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 and pushed into the corresponding task pool according to its function name stored in the task frame. In WATS, each worker tracks the execution time of the tasks executed by it. Once a task is completed, the worker updates the information of the correspondence task class.

Two types of task-generating policies, parent-first and child-first, can be adopted for generating tasks in current work-stealing schedulers. 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 Guo et al. [2009]. 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. With child-first policy, for example, the MIT Cilk uses the child-first policy, also known as work-first in Blumofe et al. [1996]. Compared to the child-first policy, tasks are generated much faster in the parent-first policy.

We have ported Cilk to spawn tasks adopting the parent-first policy since WATS tends to generate all of the tasks as soon as possible so that the history-based task allocator can allocate them to different c-groups in a short time. In addition, 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 that 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.

In order to construct the ETT of a program, for each task class, we need to collect the IPCs of its tasks on cores of all c-groups using hardware performance counters. Based on ETT, we can apply Algorithm 1 to allocate task classes to c-groups. If not all items in the ETT are determined, the task classes cannot be allocated to different c-groups appropriately. To collect the information as soon as possible, motivated by random work-stealing strategy [Frigo et al. 1998], any core c will grab a task from a random task pool when c is free. After all task classes are built, the history-based task allocator can allocate task classes to c-groups and then WATS adopts preference-based task scheduler to balance the workloads dynamically.

An interesting detail of the WATS implementation is that WATS schedules the main task of a parallel program on the fastest core, as in Saez et al. [2010a]. 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, we make all other schedulers in the experiment section launch the main task on the fastest core, although 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.

4. EVALUATION

We now evaluate WATS, including its performance over the current task schedulers, and the effectiveness of the preference-based task scheduler in WATS. Then, we evaluate the energy efficiency and scalability of WATS. After that, we also compare WATS with our previous scheduler, WATS-OLD, and discuss whether we should integrate task snatching into WATS or not. Last, we discuss other issues related to WATS.

4.1. Experiment Configurations

We use a 16-core server that has four AMD Quad-core Opteron 8380 processors (code named "Shanghai") and a 32-core server that has four Intel Octal-core Xeon X7560 processors to evaluate the performance of WATS. In the AMD Opteron 8380 processor, each core can run at 2.5GHz, 1.8GHz, 1.3GHz, and 0.8GHz. In the Intel Xeon X7560 processor, each core can run at 11 different frequencies. We adjust the frequency of each core to emulate different single-ISA AMC architectures in the experiment. To emulate AMC architectures, we use all 4 possible frequencies in the server built with

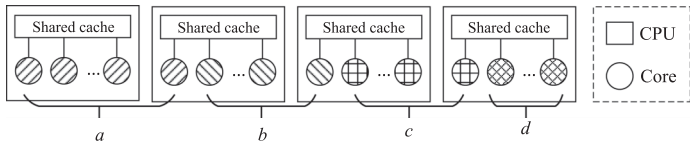


Fig. 6. Topology of the emulated AMC architectures. The numbers of cores running at four different frequencies are $a, b, c,$ and d, respectively.

Table V. The Emulated AMC Architectures in the Experiment

AMD Server	2.5GHz	1.8GHz	1.3GHz	0.8GHz
A-2-2-2-10	2	2	2	10
A-4-4-4-4	4	4	4	4
A-2-0-0-14	2	0	0	14
A-4-0-0-12	4	0	0	12
A-8-0-0-8	8	0	0	8
A-12-0-0-4	12	0	0	4
A-16-0-0-0	16	0	0	0
Intel Server	**2.262GHz**	**1.862GHz**	**1.463GHz**	**1.064GHz**
I-8-0-0-24	8	0	0	24
I-8-8-8-8	8	8	8	8
I-16-0-0-16	16	0	0	16
I-24-0-0-8	24	0	0	8
I-32-0-0-0	32	0	0	0

Table VI. Benchmarks Used in the Experiment

Name	Type	Description
BWT	Batch based	Burrows Wheeler Transform
DMC	Batch based	Dynamic Markov Coding
GA	Batch based	Island model of Genetic Algorithm [Zheng et al. 2011]
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 [Bienia et al. 2008]
Ferret	Pipeline based	Ferret from PARSEC [Bienia et al. 2008]

AMD processors and use 2.262GHz, 1.862GHz, 1.463GHz, and 1.064GHz in the server built with Intel processors.

Figure 6 provides the topology of the emulated AMC architectures. We use A-*a*-*b*-*c*-*d* to represent the emulated AMC architecture on an AMD server that has a cores running at 2.5GHz, b cores running at 1.8GHz, c cores running at 1.3GHz, and d cores running at 0.8GHz. Similarly, we use I-*a*-*b*-*c*-*d* to represent the emulated AMC architecture on an Intel server that has a cores running at 2.262GHz, b cores running at 1.862GHz, c cores running at 1.463GHz, and d cores running at 1.064GHz. Table V lists the emulated AMC architectures.

Because WATS is proposed to improve the performance of both batch and nonbatch applications with tasks that have different workloads, we use benchmarks listed in Table VI to evaluate the performance of WATS.

The source code of benchmarks are from their official Web sites [Mahoney 2013] but are adapted to run on MIT Cilk. In the batch-based benchmarks, the program launches different numbers of independent tasks (more than 128 tasks on an AMD server and 256 tasks on an Intel server) in different batches. In these benchmarks, tasks work on independent datasets of different sizes in parallel. 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 via pipelines. For each test, every benchmark is run 10 times. Because the execution time is quite stable, the average execution time is used as the result.

For pipeline-based benchmarks, WATS allocates tasks to c-groups adopting the method in Section 3.2.1. For batch-based benchmarks, WATS allocates tasks to c-groups adopting the method in Section 3.2.2.

We compare the performance of WATS with the performance of three other task schedulers: MIT Cilk, PFWS, and RTS in AMC architectures. Although MIT Cilk is originally proposed to balance fine-grained tasks [Blumofe et al. 1996], the internal work-stealing strategy is still one of the most efficient dynamic load-balancing strategies to balance coarse-grained tasks, such as tasks in batch-based programs and pipeline-based programs [Navarro et al. 2009; Mattheis et al. 2012; Maia et al. 2013].

In MIT Cilk (denoted as Cilk for short) [Blumofe et al. 1996], tasks are spawned with the child-first policy and scheduled with the traditional work-stealing policy. In PFWS (Parent-First Work Stealing) [Guo et al. 2009], parallel tasks are spawned with the parent-first policy and scheduled with the traditional work-stealing policy. In RTS (Random Task Snatching) [Bender and Rabin 2000], 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. The snatch operation is implemented by swapping the two threads on the faster core and the slower core.

To evaluate the effectiveness of the *help-the-weaker-first* policy, we also compare the performance of WATS with WATS-NP, a scheduler that adopts the history-based task allocator, but cores in one c-group are not allowed to obtain 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 allocator. To ensure fairness of comparison, WATS, PFWS, RTS, and WATS-NP are implemented by modifying MIT Cilk.

We also emulate an AMC architecture that consists of four Out-of-Order Xeon cores and four In-Order Atom cores using a Marss-86 [Patel et al. 2011] simulator and evaluate the performance of WATS on it. However, although the benchmarks in Table VI can run successfully in this emulated architecture, the simulator is too slow for us to collect the execution time of all benchmarks. From the successfully collected experimental result of GA and Dedup with a very small work set on this architecture, we find that WATS greatly outperforms all other schedulers. Therefore, to collect experimental results in reasonable time, we mainly use the fast architectures in Table V to evaluate the performance of WATS.

4.2. Performance of WATS

We have tested the performance of the benchmarks in all of the $6 + 4 = 10$ AMC architectures and $1 + 1 = 2$ symmetric multicore architectures. Figure 7 presents the performance of the batch-based benchmarks in all of the emulated AMC architectures, and Figure 8 presents the performance of the pipeline-based benchmarks in all of the emulated AMC architectures. In Figures 7 and 8, the y axis on the left is the execution time of the benchmarks in AMC emulated on the AMD server, and the y axis on the right is the execution time of the benchmarks in AMC emulated on the Intel server.

From Figure 7, we can find that WATS can significantly improve the performance of the batch-based benchmarks in all the emulated AMC architectures. In A-2-2-2-10 to A-12-0-0-4 that are emulated on the AMD server, WATS improves the performance of batch-based benchmarks, with the performance gains ranging from 10.7% to 66.1% compared to Cilk and PFWS, and with performance gains ranging from 2.4% to 65.6% compared to RTS. In I-8-0-0-24 to I-24-0-0-8 that are emulated on the Intel server, WATS improves the performance of batch-based benchmarks, with the performance

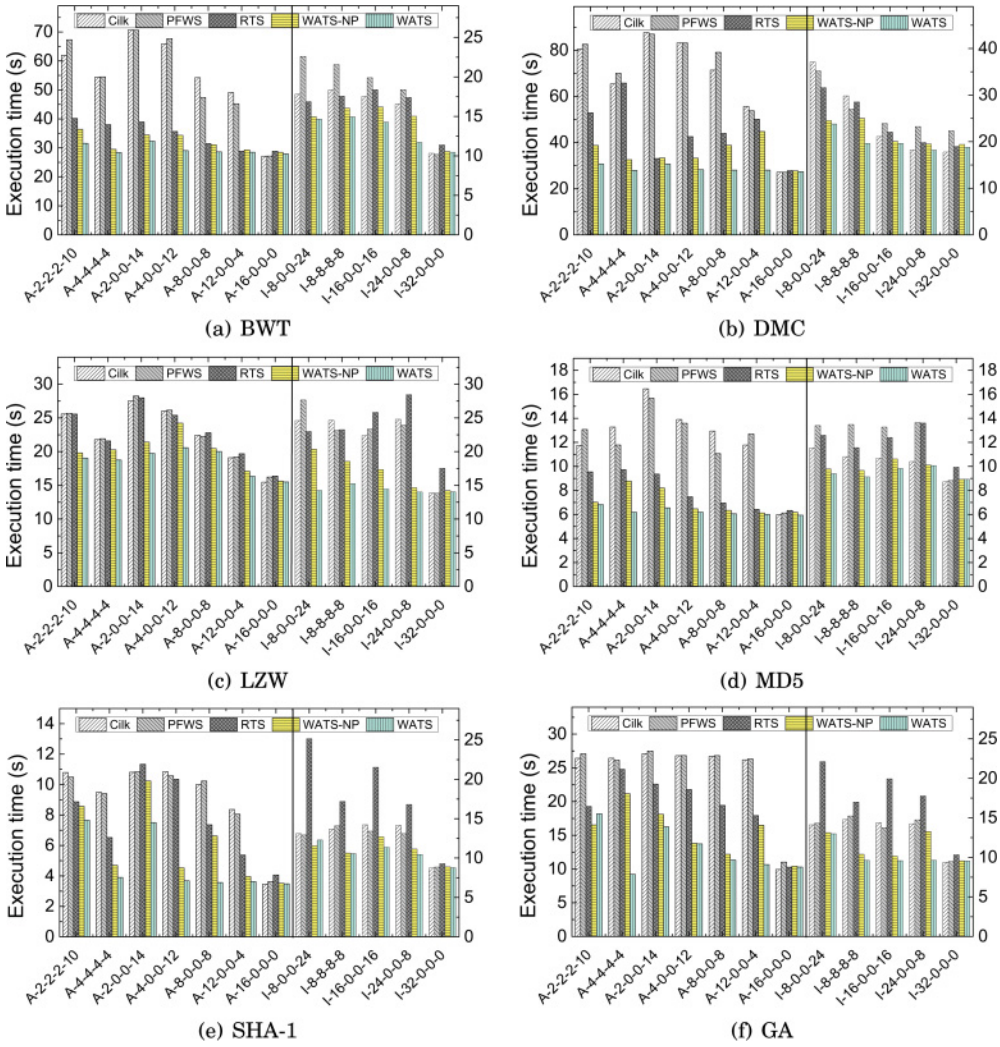


Fig. 7. Performance of the batch-based benchmarks in the emulated AMC architectures.

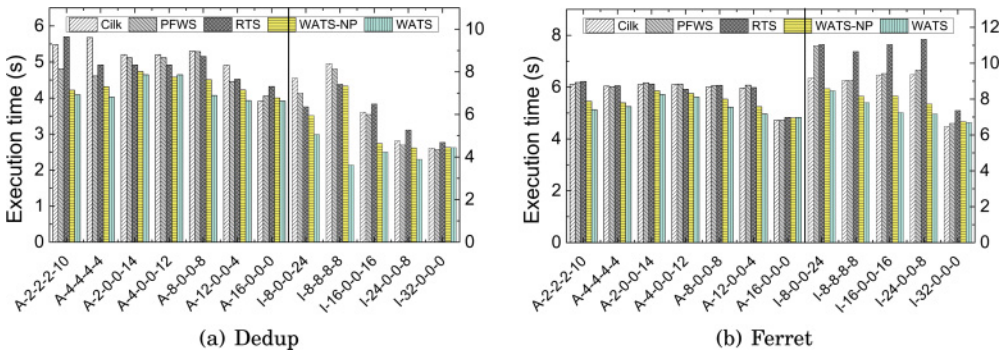


Fig. 8. Performance of the pipeline-based benchmarks in the emulated AMC architectures.

gains ranging from 3.3% to 38.1% compared to Cilk and PFWS, and with the performance gains ranging from 4.2% to 51.1% compared to RTS. For example, for SHA-1 in A-4-0-0-12 in Figure 7(e), WATS reduces the execution time up to 66.1% compared to Cilk. For LZW in I-8-8-8-8 in Figure 7(c), WATS reduces the execution time up to 38.1% compared to Cilk.

Figure 8 shows that WATS can significantly improve the performance of the pipeline-based benchmarks in all of the AMC architectures. In A-2-2-2-10 to A-12-0-0-4 , WATS improves the performance of pipeline-based benchmarks, with the performance gains up to 29.1% compared to Cilk and PFWS, and with performance gains up to 28.2% compared to RTS. In I-8-0-0-24 to I-24-0-0-8, WATS improves the performance of pipeline-based benchmarks, with the performance gains up to 44.9% compared to Cilk and PFWS, and with the performance gains up to 36.8% compared to RTS.

Careful readers may find that the performance of WATS in this article is similar to the performance of the scheduler in our previous conference paper [Chen et al. 2012a]. However, the scheduler in Chen et al. [2012a] cannot perfectly schedule the batch-based benchmarks in Figure 7, in which the percentage of tasks executing the same function among all tasks is not same in different batches. WATS in this article has removed this limitation by generating all of the tasks in a batch first and then allocating task classes to c-groups based on the real workloads. We will discuss this issue in Section 4.6. In addition, our scheduler in Chen et al. [2012a] only works for CPU-bound applications. By collecting the execution time of every task class on cores in every c-group, WATS in this article has removed this limitation as well.

The good performance of WATS comes from its balanced workloads in the AMC architectures. With the history-based task allocator, WATS allocates tasks with a heavy workload to fast cores and tasks with a 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 scheduler.

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

From Figure 7, we can find that WATS performs better for batch-based programs in the AMCs emulated on the AMD server than the AMCs emulated on the Intel server. The better performance on the AMD server comes from the large gap between the fast core speed and the slow core speed. As shown in Table V, the speed of the slowest cores is only $\frac{0.8}{2.5} = 32\%$ of the speed of the fastest cores in A-2-2-2-10 to A-12-0-0-4, whereas the speed of the slowest cores is $\frac{1.064}{2.262} = 47\%$ of the speed of the fastest cores in I-8-0-0-24 to I-24-0-0-8. A task with heavy workload is slowed down by more times in A-2-2-2-10 to A-12-0-0-4 if the task is scheduled to a slow core. Therefore, the larger the gap between the fastest cores and the slowest cores in an AMC architecture, the more WATS can improve the performance of applications that have tasks with different workloads.

Compared to Cilk and PFWS, RTS can also improve the performance of most benchmarks in AMC architectures. 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 PFWS, for most benchmarks, RTS improves the performance of the benchmarks up to 60.9% in A-2-2-2-10 to A-12-0-0-4 and up to 35.6% in I-8-0-0-24 to I-24-0-0-8.

However, for many other benchmarks, such as GA in A-2-0-0-14 and MD5 in I-24-0-0-8, RTS even degrades the performance of the benchmarks due to the overheads that come from the frequent task snatching (or context switching). In addition, since RTS is not aware of the workloads of the tasks, it is possible for faster cores to snatch

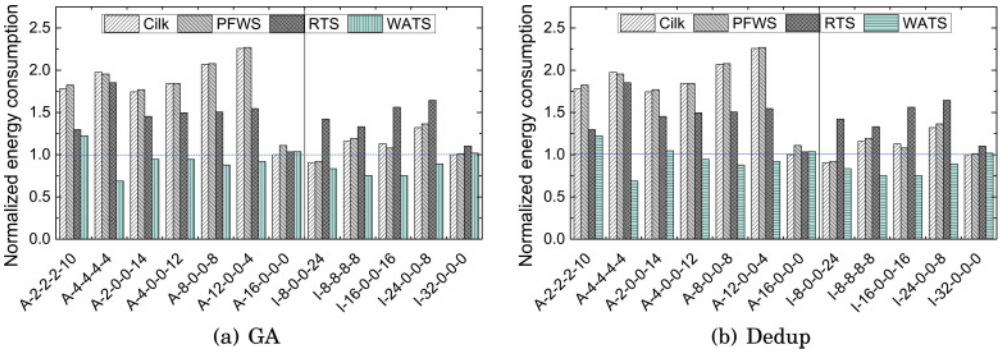


Fig. 9. Normalized energy consumption of GA and Dedup in Cilk, PFWS, RTS, and WATS.

tasks with light workload, in which case the makespan cannot be reduced. Especially in AMC emulated in an Intel server, there are overall 32 cores. The large number of cores often lead to more task-snatching operations in RTS. Therefore, RTS performs poorly in I-8-0-0-24 to I-24-0-0-8, and it performs much worse than WATS.

For symmetric multicore architectures, WATS schedules tasks similarly to PFWS. Therefore, as shown in Figures 7 and 8, WATS performs the same as PFWS on A-16-0-0-0 and I-32-0-0-0. The overhead in WATS is negligible compared with traditional work stealing in symmetric architecture.

Figures 7 and 8 also show that WATS can adapt to different AMC architectures and improve performance of benchmarks automatically. In addition, when an AMC architecture has a small number of fast cores (e.g., eight fast cores in I-8-0-0-24, two fast cores in A-2-0-0-14), the frequent context switching on fast cores that comes from task snatching reduces the computing time of fast cores on tasks. In this case, RTS degrades the overall performance of some benchmarks (e.g., GA) compared with Cilk and PFWS.

4.3. Effectiveness of the Preference-Based Task Scheduler

As shown in Figures 7 and 8, the performance of WATS is always better than the performance of WATS-NP. Especially for Dedup in I-8-8-8-8, WATS-NP even prolongs the execution time of Dedup up to 32.3% compared to WATS. For GA in A-4-4-4-4, WATS-NP even prolongs the execution time of GA up to 45.8% compared to WATS. The preference-based task scheduler in WATS is very helpful when handling slightly unbalanced workloads. Since the history-based task allocator may mis-allocate the tasks to the wrong c-groups due to its static approximation of the workloads of dynamic tasks, the preference-based task scheduler can remedy this imprecision.

It is worth noting that the history-based task allocator has mostly done effective allocation of tasks according to Figures 7 and 8. WATS-NP performs better than Cilk and PFWS, which means that the history-based allocation algorithm is more effective than random task stealing in terms of load balancing in AMC architecture.

4.4. Energy Efficiency of WATS

Figure 9 evaluates the energy efficiency of WATS. It gives energy consumption of the batch-based benchmark GA and the pipeline-based benchmark Dedup, although other benchmarks show similar results.

In Figure 9, the energy consumption of GA and Dedup in AMCs emulated on the AMD server is normalized against their energy consumption in Cilk in A-16-0-0-0.

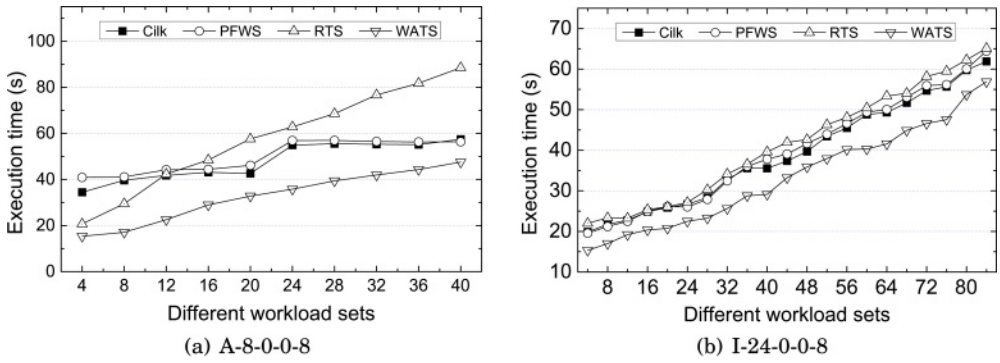


Fig. 10. Performance of GA with different workloads in A-8-0-0-8 and I-8-0-0-24.

The energy consumption of GA and Dedup in AMCs emulated on the Intel server is normalized against their energy consumption in Cilk in I-32-0-0-0.

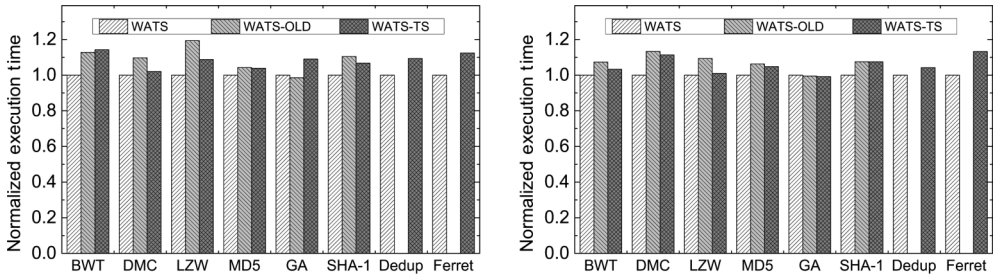
In Figure 9, we find that WATS can always reduce the energy consumption of the benchmarks compared with Cilk, PFWS, and RTS in all of the emulated AMC architectures. However, the energy consumption of the benchmarks is increased in some AMC architectures (e.g., A-2-2-2-10) compared with their energy consumption in symmetric multicore architectures (e.g., A-16-0-0-0), although the cores that run at lower frequencies consume less energy. The increased energy consumption results from the increased execution time of the benchmarks in AMC architectures.

If the frequencies of all cores are fixed (one assumption of AMC architectures) and the frequencies of the cores are not proper for an application, the energy consumption of executing the application can be increased in AMC architectures. Although we have implemented a system for saving power based on DVFS and WATS through adjusting the frequencies of cores dynamically for different applications at runtime, we do not present the approach here due to the page limitation. The techniques proposed in Ghiasi et al. [2005] and Shelepov et al. [2009] can also be used to reduce energy of executing memory-intensive programs in AMC architectures.

4.5. Scalability of WATS

Figure 10 evaluates the scalability of WATS. It gives the performance of GA under different distributions of workloads in A-8-0-0-8 and I-24-0-0-8, although other benchmarks show similar results in various AMC architectures. In the AMC emulated on an AMD server that has 16 cores, GA launches 128 tasks with four different workloads in each batch. In the AMC emulated on an Intel server that has 32 cores, GA launches 256 tasks with four different workloads in each batch. The number of tasks with each type of workload is adjusted to evaluate the scalability of WATS when the number of tasks with heavy workload increases. In Figure 10(a), the distribution of workloads from high to low follows the pattern $\alpha, \alpha, \alpha, 128 - 3\alpha$. In Figure 10(b), the distribution of workloads from high to low follows the pattern $\alpha, \alpha, \alpha, 256 - 3\alpha$. In both parts of the figure, α is adjusted as shown by the x axis in the figure. In Figure 10, the fastest core needs less than 200 microseconds to process the tasks with the lowest workload and needs less than 10 milliseconds to process the tasks with the highest workload.

From the figure, we can see that WATS works fine under different distributions of workloads. In A-8-0-0-8, when α is small and the workloads are mostly light, WATS reduces the GA execution time by 55.4% compared to Cilk. When α is large and the workloads are mostly heavy, WATS can still reduce the execution time by 17.2% compared to Cilk. In I-24-0-0-8, when α is small, WATS reduces the GA execution time by



(a) A-4-0-0-12 (b) I-8-8-8-8

Fig. 11. Performance of the benchmarks in WATS, WATS-OLD, and WATS-TS.

22.8% compared to Cilk. When α is large and the workloads are mostly heavy, WATS can still reduce the execution time by 8.1% compared to Cilk. Therefore, WATS is scalable with and can adapt to different workloads.

However, in A-8-0-0-8, shown in Figure 10(a), RTS does not work well when the workloads are mostly heavy (e.g., α is 20), as it even degrades the performance of GA by 54.1% compared to Cilk and PFWS. This is because fast cores are not able to snatch all of the heavy tasks that are allocated to the slow cores when there are too many heavy tasks. Moreover, the computing ability of fast cores is wasted at frequent context switching when the workloads are mostly heavy. This result again supports our philosophy of WATS that an optimal task allocation is more important than rescuing policies such as task snatching.

In I-24-0-0-8, shown in Figure 10(b), RTS works even worse than Cilk and PFWS for all workloads, as it degrades the performance up to 10.7% compared to Cilk. In I-24-0-0-8, the reduced execution time of GA that originates from task snatching in RTS is small because the difference between the speed of fast cores and the speed of slow cores is small. It is quite possible that the reduced execution time is smaller than the increased execution time that originates from the context switching in RTS. Therefore, if the gap between the speed of fast cores and the speed of slow cores is small, the performance of RTS is poor.

4.6. Effectiveness of WATS for Irregular Batch-Based Programs

As presented in Section 3.2, for any batch in a batch-based program, WATS let the program generate all of the tasks in the batch first and then allocates the real task classes created in the batch to different c-groups. To evaluate this newly proposed strategy in WATS for irregular batch-based programs, we compare the performance of WATS with WATS-OLD, a scheduler that allocates tasks in a batch to c-groups totally based on the allocation of tasks completed in history as in Section 3.2.1 for nonbatch programs and in our previous conference paper [Chen et al. 2012a].

In this experiment, for the batch-based benchmarks except benchmark GA, we randomly choose datasets for all of the tasks, and therefore the percentage of tasks executing the same function among all tasks is not the same in different batches. For GA, the presented constraint is still obeyed due to the algorithm limitation.

Figure 11 shows the performance of the batch-based benchmarks in WATS and WATS-OLD. We can find that the batch-based programs always perform better in WATS than in WATS-OLD. In WATS, because all tasks in a batch are spawned first, WATS can get the real workload of every task class and thus can allocate the tasks to c-groups near optimally. However, in WATS-OLD, because the near-optimal allocation of tasks completed in history is not near optimal for future tasks anymore, the nonoptimal allocation degrades the performance of batch-based programs in WATS-OLD. For

GA, because the percentage of tasks executing the same function among all tasks is the same in different batches, the tasks are allocated to c-groups in the same way in WATS and WATS-OLD. Therefore, GA performs similarly in WATS and WATS-OLD. The slightly worse performance of GA in WATS comes from its slightly heavier overhead in collecting execution time of every task class in every c-group. Note that WATS-OLD still performs much better than Cilk, PFWS, and RTS because the preference-based task scheduler can handle slightly unbalanced workloads dynamically.

4.7. Integrating Task Snatching in WATS

It is also of interest to discover whether or not task snatching is also effective in WATS and thus should be integrated into WATS. To investigate this issue, we implemented the scheduler WATS-TS, where fast cores snatch tasks from slow cores when the fast cores cannot obtain any tasks using the preference-based task scheduling 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 2. 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.

Figure 11 also shows the performance of the benchmarks in WATS and WATS-TS in A-4-0-0-12 and I-8-8-8-8. From the figure, we surprisingly see that the performance of WATS-TS is slightly worse than WATS. Especially for BWT and Ferret in A-4-0-0-12 and DMC and Ferret in I-8-8-8-8, WATS-TS increases the execution time up to 14.2% compared to WATS.

Figure 11 tells us that WATS has satisfactorily balanced the workloads in AMC architectures. 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 task snatching.

4.8. Discussion

Not surprisingly, WATS has one main limitation. If most tasks in an 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, recursive divide-and-conquer programs such as *nqueens* and *fib* 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 work stealing is used instead to schedule the program. Furthermore, if the program is also memory intensive, our previous schedulers [Chen et al. 2011, 2012b, 2013a] can be adopted to improve its performance by reducing the cache misses. Therefore, the limitation presented will not affect the applicability of WATS since the compiler can identify the class of programs that are suitable for WATS.

5. RELATED WORK

Researchers have shown that the AMC architectures can achieve high performance and low power consumption [Kumar et al. 2004, 2005; Balakrishnan et al. 2005; Hill and Marty 2008]. 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 work 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 architectures.

Many studies have been done to explore optimal task scheduling in different parallel platforms [Chen et al. 2013b; Shelepov et al. 2009]. Especially in AMC, many studies on scheduling focus on resource allocation at the OS level [De Vuyst et al. 2006; Rosenberg and Chiang 2010; Bhadauria and McKee 2010; Li et al. 2007]. They aim to achieve high system throughput by balancing the hardware resources (e.g., cores and caches) among different programs. The CAMP [Saez et al. 2010b] OS-level scheduler is proposed to optimize system throughput by devoting fast cores to run high-speedup applications in AMC. Because tasks in the same task-based programs can often achieve similar speedup ratios on fast cores, CAMP is not applicable in the targeted scenario of WATS. In El-Moursy et al. [2006], 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 Lakshminarayana et al. [2009], age-based scheduling is proposed to schedule the threads with larger remaining time to fast cores. In Koufaty et al. [2010], the authors propose a bias scheduling that matches threads to the right type of cores through dynamically monitoring the bias of the threads in order to maximize the system throughput. The studies presented 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 Suleman et al. [2009], Accelerated Critical Sections (ACS) is proposed to accelerate the execution of critical sections by migrating the threads with critical sections to fast cores. Similar to ACS, in Joao et al. [2012], a cooperative software-hardware mechanism, Bottleneck Identification and Scheduling (BIS) is proposed to identify and accelerate the most critical bottlenecks. BIS identifies the most critical bottlenecks by measuring the number of cycles that threads have to wait for each bottleneck and accelerates the bottlenecks using fast cores on an AMC architecture. In addition, although BIS needed to add some structures in hardware, WATS is proposed based on a pure software approach. In Hofmeyr et al. [2010], 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 [Bender and Rabin 2000] (i.e., RTS in Section 4.2), although it addresses the problem in the context of an Asymmetric Multiprocessor (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 previously, RTS cannot balance tasks as well as WATS due to its lack of workload information about the tasks.

Work stealing has been extensively studied and adopted by parallel programming environments [Blumofe et al. 1996; Reinders 2007; Guo et al. 2009, 2010], although it does not perform well in AMC. An extension to task stealing for improving cache performance in multicore architectures has recently been proposed [Chen et al. 2012b]. The preference-based work-stealing policy in WATS is a novel extension to task stealing to balance workloads among different groups of cores in AMC.

If the number of tasks and the workloads of tasks in the same task classes are totally repeatable and can be estimated accurately, similar to our history-based task allocation, some other task-allocating algorithms [Miguet and Pierson 1997; Hochbaum and Shmoys 1988] can provide a near-optimal scheduling. However, for many real applications, the workloads of tasks are not totally repeatable. As far as we know, the linear programming–based technique cannot tolerate the nonrepeatability due to its static scheduling. On the contrary, WATS can tolerate some nonrepeatability of the estimation of the tasks due to the preference-based task scheduler. WATS uses the preference-based task scheduler to further balance the workloads when the tasks were poorly assigned due to the nonrepeatability of tasks.

In our previous conference paper [Chen et al. 2012a], the scheduler only works when the percentage of tasks executing the same function among all tasks is almost the same during the execution of a parallel application. This assumption is strong and is not always true in real parallel applications. Enhanced from Chen et al. [2012a], WATS in this article does not rely on the strong assumption anymore since the number of tasks in each task classes is known in the history-based task allocator. In addition, we have evaluated the improved WATS on one more multicore server and have produced more interesting results based on the additional experiment.

6. CONTRIBUTIONS AND CONCLUSIONS

The contributions of this article are as follows:

—We have identified, defined, and formalized the problem of unbalanced workloads in AMC architectures and have given theoretical guidance to optimal task allocation in AMC architectures.
—We have proposed a history-based task allocator that can allocate tasks in single-ISA AMC architectures near optimally.
—We have proposed a novel preference-based task scheduler that can effectively balance workloads among different groups of cores.
—Based on the techniques presented, we have implemented WATS, which achieves a performance gain of up to 66.1% compared to the random work-stealing approach commonly employed.

Single-ISA AMCs are promising due to their high performance and power efficiency. It is essential for parallel applications to run on AMC architectures efficiently. Although task-scheduling policies such as work 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 nonoptimal allocation and that static task allocation can produce near-optimal allocation if the workloads of the tasks are known. Therefore, we propose a history-based task allocator that takes advantage of the static allocation by using the historical statistics of the tasks to predict the execution time of future tasks on cores in different c-groups. From our experiments, we showed that the history-based task allocator can produce appropriate allocation and that its extra overhead is small.

For any occasional inaccurate or incorrect allocation of tasks, the preference-based task scheduler comes to play. It can remedy any slightly unbalanced allocation and effectively schedule tasks among c-groups. The experimental results show that WATS is effective and that our approach to the scheduling problem in single-ISA AMC is valid.

REFERENCES

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

S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai. 2005. The impact of performance asymmetry in emerging multicore architectures. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*. IEEE, 506–517.

M. A. Bender and M. O. Rabin. 2000. Scheduling Cilk multithreaded parallel programs on processors of different speeds. In *Proceedings of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures*. ACM, 13–21.

M. Bhadauria and S. A. McKee. 2010. An approach to resource-aware co-scheduling for cmps. In *Proceedings of the 24th ACM International Conference on Supercomputing*. ACM, 189–199.

C. Bienia, S. Kumar, J. P. Singh, and K. Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. ACM, 72–81.

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

Q. Chen, Y. Chen, Z. Huang, and M. Guo. 2012a. WATS: Workload-Aware Task Scheduling in asymmetric multi-core architectures. In *Proceedings of the 26th International Parallel and Distributed Processing Symposium*. IEEE, 249–260.

Q. Chen, M. Guo, Q. Deng, L. Zheng, S. Guo, and Y. Shen. 2013b. HAT: History-based auto-tuning MapReduce in heterogeneous environments. *Journal of Supercomputing*, 1–17.

Q. Chen, M. Guo, and Z. Huang. 2012b. CATS: Cache Aware Task-Stealing based on online profiling in multi-socket multi-core architectures. In *Proceedings of the 26th International Conference on Supercomputing*. IEEE, 163–172.

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

Q. Chen, Z. Huang, M. Guo, and J. Zhou. 2011. CAB: Cache-Aware Bi-tier task-stealing in multi-socket multi-core architecture. In *Proceedings of the International Conference on Parallel Processing*. IEEE.

M. De Vuyst, R. Kumar, and D. M. Tullsen. 2006. Exploiting unbalanced thread scheduling for energy and performance on a CMP of SMT processors. In *Proceedings of the International Parallel and Distributed Processing Symposium*. IEEE.

A. El-Moursy, R. Garg, D. H. Albonesi, and S. Dwarkadas. 2006. Compatible phase co-scheduling on a CMP of multi-threaded processors. In *Proceedings of the International Parallel and Distributed Processing Symposium*. IEEE.

M. Frigo, C. E. Leiserson, and K. H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 212–223.

S. Ghiasi, T. Keller, and F. Rawson. 2005. Scheduling for heterogeneous processors in server systems. In *Proceedings of the 2nd Conference on Computing Frontiers*. ACM, 199–210.

Y. Guo, R. Barik, R. Raman, and V. Sarkar. 2009. Work-first and help-first scheduling policies for async-finish task parallelism. In *Proceedings of the International Parallel and Distributed Processing Symposium*. IEEE.

Y. Guo, J. Zhao, V. Cave, and V. Sarkar. 2010. SLAW: A scalable locality-aware adaptive work-stealing scheduler. In *Proceedings of the International Parallel and Distributed Processing Symposium*. IEEE.

M. D. Hill and M. R. Marty. 2008. Amdahl's law in the multicore era. *Computer* 41, 7, 33–38.

D. S. Hochbaum and D. B. Shmoys. 1988. A polynomial approximation scheme for scheduling on uniform processors: Using the dual approximation approach. *SIAM Journal on Computing* 17, 539.

S. Hofmeyr, C. Iancu, and F. Blagojević. 2010. Load balancing on speed. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 147–158.

J. A. Joao, M. Aater Suleman, O. Mutlu, and Y. N. Patt. 2012. Bottleneck identification and scheduling in multithreaded applications. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*. 223–234.

D. Koufaty, D. Reddy, and S. Hahn. 2010. Bias scheduling in heterogeneous multi-core architectures. In *Proceedings of the 5th European Conference on Computer Systems*. ACM, 125–138.

R. Kumar, D. M. Tullsen, N. P. Jouppi, and P. Ranganathan. 2005. Heterogeneous chip multiprocessors. *Computer* 38, 11, 32–38.

R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas. 2004. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*. IEEE.

N. B. Lakshminarayana, J. Lee, and H. Kim. 2009. Age based scheduling for asymmetric multiprocessors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. ACM, 25.

J. K. Lee and J. Palsberg. 2010. Featherweight X10: A core calculus for async-finish parallelism. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM, 25–36.

T. Li, D. Baumberger, D. A. Koufaty, and S. Hahn. 2007. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Proceedings of the ACM/IEEE Conference on Supercomputing*. ACM.

J. W. S. Liu and C. L. Liu. 1974. *Bounds on Scheduling Algorithms for Heterogeneous Computing Systems*. Department of Computer Science, University of Illinois at Urbana–Champaign.

M. Mahoney. 2013. *Data Compression Programs*. http://mattmahoney.net/dc/.

C. Maia, L. Nogueira, and L. M. Pinho. 2013. Scheduling parallel real-time tasks using a fixed-priority work-stealing algorithm on multiprocessors. In *Proceedings of the 8th International Symposium on Industrial Embedded Systems*. IEEE.

S. Mattheis, T. Schuele, A. Raabe, T. Henties, and U. Gleim. 2012. Work stealing strategies for parallel stream processing in soft real-time systems. In *Architecture of Computing Systems*. Springer, 172–183.

S. Miguet and J.-M. Pierson. 1997. Heuristics for 1D rectilinear partitioning as a low cost and high quality answer to dynamic load balancing. In *Proceedings of HPCN Europe*. 550–564.

A. Navarro, R. Asenjo, S. Tabik, and C. Caşcaval. 2009. Load balancing using work-stealing for pipeline parallelism in emerging applications. In *Proceedings of the 23rd International Conference on Supercomputing*. ACM, 517–518.

A. Patel, F. Afram, S. Chen, and K. Ghose. 2011. MARSS: A full system simulator for multicore x86 CPUs. In *Proceedings of the 48th Design Automation Conference*. ACM, 1050–1055.

J. Reinders. 2007. *Intel Threading Building Blocks*. O'Reilly.

A. L. Rosenberg and R. C. Chiang. 2010. Toward understanding heterogeneity in computing. In *Proceedings of the International Parallel and Distributed Processing Symposium*. IEEE, 1–10.

J. C. Saez, A. Fedorova, M. Prieto, and H. Vegas. 2010a. Operating system support for mitigating software scalability bottlenecks on asymmetric multicore processors. In *Proceedings of the 7th ACM International Conference on Computing Frontiers*. ACM, 31–40.

J. C. Saez, M. Prieto, A. Fedorova, and S. Blagodurov. 2010b. A comprehensive scheduler for asymmetric multicore systems. In *Proceedings of the 5th European Conference on Computer Systems*. ACM, 139–152.

D. Shelepov, J. C. Saez Alcaide, S. Jeffery, A. Fedorova, N. Perez, Z. F. Huang, S. Blagodurov, and V. Kumar. 2009. HASS: A scheduler for heterogeneous multicore systems. *ACM SIGOPS Operating Systems Review* 43, 2, 66–75.

M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt. 2009. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 253–264.

K. Van Craeynest, A. Jaleel, L. Eeckhout, P. Narvaez, and J. Emer. 2012. Scheduling heterogeneous multi-cores through performance impact estimation (PIE). In *Proceedings of the 39th International Symposium on Computer Architecture*. IEEE, 213–224.

L. Zheng, Y. Lu, M. Ding, Y. Shen, M. Guo, and S. Guo. 2011. Architecture-based performance evaluation of genetic algorithms on multi/many-core systems. In *Proceedings of the 14th International Conference on Computational Science and Engineering*. IEEE, 321–334.

Received June 2013; revised November 2013; accepted November 2013