CATS: Cache Aware Task-Stealing based on Online Profiling in Multi-socket Multi-core Architectures

Quan Chen Minyi Guo
Shanghai Key Laboratory of Scalable Computing and Systems,
Department of Computer Science and Engineering,
Shanghai Jiao Tong University, China
chen-quan@sjtu.edu.cn, guo-my@cs.sjtu.edu.cn

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

Abstract

Multi-socket Multi-core architectures with shared caches in each socket have become mainstream when a single multi-core chip cannot provide enough computing capacity for high performance computing. However, traditional task-stealing schedulers tend to pollute the shared cache and incur severe cache misses due to their randomness in stealing. To address the problem, this paper proposes a Cache Aware Task-Stealing (CATS) scheduler, which uses the shared cache efficiently with an online profiling method and schedules tasks with shared data to the same socket. CATS adopts an online DAG partitioner based on the profiling information to ensure tasks with shared data can efficiently utilize the shared cache. One outstanding novelty of CATS is that it does not require any extra user-provided information. Experimental results show that CATS can improve the performance of memory-bound programs up to 74.4% compared with the traditional task-stealing scheduler.

Keywords Cache Aware, Task-stealing, Online Profiling, Multi-socket Multi-core, Cache misses

1. Introduction

Multi-core processors have become mainstream since they have better performance per watt and larger computational capacity than complex single-core processors. However, each single CPU die can hardly contain too many cores (such as, more than 128 cores) due to the physical limitations in industrial manufacture. To fulfill the urgent desire on powerful computers, many multi-core processors are integrated together into a Multi-socket Multi-core (MSMC) architecture. In an MSMC architecture, each CPU die is plugged into a socket and the cores in the same socket have a shared cache; however, the cores from different sockets can only share the main memory.

To fully utilize the MSMC architectures, many parallel programming environments have been proposed. In some parallel programming environments, such as Pthread [9], MPI [18] and Maotai [34], parallelism is expressed through multithreading. Programmers need to launch threads and assign tasks to these threads manually in multithreading. However, the manual assignment of tasks is often burdensome for developing applications. To relieve the burden of parallelization and task assignment, parallel programming environments, such as Cilk [8], Cilk++ [25], TBB [30], Java's fork-join framework [23], X10 [24], and OpenMP [2], assign and schedule tasks automatically. *Task-sharing* [2] and *task-stealing* (also known as work-stealing[1]) [7] are the two most famous task scheduling strategies.

In task-sharing, workers (i.e. threads) push new tasks into a central task pool when they are generated. Tasks are popped out from the task pool when workers are free to execute them. The push and pop operations need to lock the central task pool, which often causes serious lock contention.

Task-stealing, on the other hand, provides an individual task pool for each worker. Most often each worker pushes tasks to and pops tasks from its own task pool without locking. Only when a worker's task pool is empty, it tries to steal tasks from other workers with locking. Since there are multiple task pools for stealing, the lock contention is much lower than task-sharing even at task steals. Therefore, task-stealing performs better than task-sharing as the number of workers increases.

However, both task-sharing and task-stealing strategies schedule tasks randomly to different cores. This randomness can cause shared cache misses and degrade the performance of memory-bound applications on MSMC architectures (to be discussed in detail in Section 2). For example, two tasks with shared data may be allocated to different sockets due to the randomness in these strategies. In this case, both tasks cannot share the data loaded to the shared cache but have to read the shared data from the main memory which could be hundreds times slower than the shared cache. If the two tasks are scheduled to cores in the same socket, only one of them needs to read the shared data from the main memory while the other task can access the shared data from the shared cache directly.

Based on this observation, this paper proposes a Cache Aware Task-Stealing (CATS) scheduler that automatically schedules tasks with shared data into the same socket. CATS consists of two parts: an *online DAG partitioner* and a *bi-tier task-stealing scheduler*. The online DAG partitioner automatically divides the execution DAG (Directed Acyclic Graph) of a parallel program into the inter-socket tier and the intra-socket tier based on the profiling information collected during execution. The bi-tier task-stealing scheduler allows tasks in the inter-socket tier to be stolen across sockets, while tasks in the intra-socket tier are scheduled within the same socket. Since tasks from the intra-socket tier often share data, CATS can use the shared cache efficiently.

[1] we use "task-stealing" in this paper for the consistency of terms.

The contributions of this paper are as follows.

- We propose an online profiling method that automatically collects run-time profiling information for cache aware task scheduling. It enables the task scheduler to optimally utilize the shared cache without extra user-provided information.
- We propose, in CATS, an online DAG partitioner that optimally divides tasks into the inter-socket tier and the intra-socket tier based on the profiling information, and a bi-tier task-stealing algorithm that schedules tasks with shared data to the same socket.
- We demonstrate that CATS significantly reduces the shared cache misses and thus improves the performance of memory-bound applications. The experiment shows that CATS can achieve a performance gain of up to 74.4% for memory-bound applications.

The rest of this paper is organized as follows. Section 2 describes the problem and explains the motivation of CATS. Section 3 presents CATS, including the online DAG partitioner and the bi-tier task-stealing scheduler. Section 4 shows the experimental results, performance evaluation and the limitations of CATS. Section 5 discusses the related work. Section 6 draws conclusions and sheds light on future work.

2. Problem and Motivation

For many parallel programming environments such as Cilk, the execution of a parallel program can often be expressed by a Directed Acyclic Graph (DAG) $G = (V, E)$, where V is a set of nodes, and E is a set of directed edges [17]. Each node in a DAG represents a task (i.e., a set of instructions) that must be executed sequentially without preemption, and the edges in a DAG correspond to the dependence relationship among the nodes. Fig. 1 shows execution DAG of a general parallel program. In the figure, the solid lines represent the task generating relationship and the strings by the side of nodes are the identifiers of the corresponding tasks.

2.1 The problem

We use Fig. 1 as an example to explain the problem of shared cache pollution in an MSMC architecture. In many parallel programs based on the *Jacobi iteration algorithm*, neighbor tasks need to access some shared data. For example, *Five-point heat distribution* and *Successive Over-Relaxation* are examples of such parallel programs. Therefore, γ_1 and γ_2, γ_3 and γ_4 in Fig. 1 have shared data respectively.

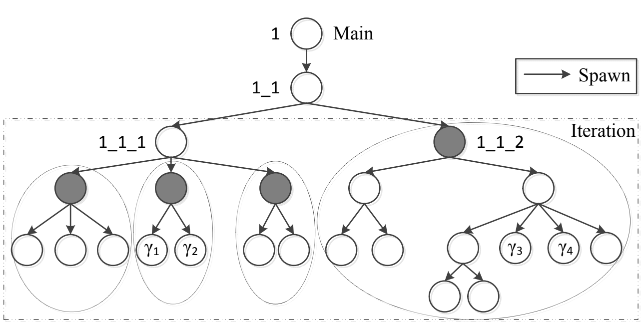
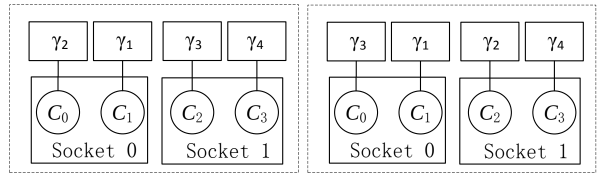


Figure 1. A general execution DAG for iteration-based parallel programs.

We assume the parallel program in Fig. 1 runs on a dual-socket dual-core architecture. If γ_1, γ_2, γ_3 and γ_4 are scheduled as shown in Fig. 2(a), the shared data between γ_1 and γ_2 and the shared data between γ_3 and γ_4 is only read into the shared cache once from the main memory. Since most tasks can access the shared data in the shared cache of the socket, cache misses are reduced in this schedule.



(a) Optimal scheduling (b) Another possible scheduling

Figure 2. Two possible scheduling of γ_1, γ_2, γ_3 and γ_4 on a dual-socket dual-core architecture. The first scheduling can gain performance improvement due to cache sharing and reduction of memory footprint.

However, for random task-stealing, since it randomly chooses a victim to steal tasks, γ_1, γ_2, γ_3 and γ_4 are likely to be randomly scheduled to the cores as shown in Fig. 2(b). In this case, each task needs to read all its data from the main memory. This larger memory footprint leads to more compulsory cache misses. Even worse, if the memory footprint exceeds the capacity of the shared cache, the situation leads to more capacity cache misses and increases the chances of conflict cache misses. The resulted larger number of cache misses will lead to the worse performance of memory-bound applications.

Though there were several task schedulers proposed [4, 5] to reduce cache misses, they either need extra user-provided information [11], or are not general enough for MSMC architectures [4].

2.2 Proposed Solution

If a task-stealing scheduler can ensure tasks with shared data are scheduled to the same socket as shown in Fig. 2(a), the shared cache misses will be minimized and the performance of memory-bound applications can be improved. To achieve the purpose, we propose the *Cache Aware Task-Stealing* (CATS) scheduler in this paper.

CATS is proposed based on the following three observations of the execution of parallel programs as shown in Fig. 1. First, parallel tasks create child tasks recursively until the data set for each leaf task is small enough. During the procedure, only the leaf tasks physically touch the data. Second, a parallel program often works on the same data set for a large number of iterations. Finally, neighbor tasks usually share some data.

Based on the runtime profiling information, CATS can divide an execution DAG into the inter-socket tier and the intra-socket tier. For example, CATS may divide the execution DAG in Fig. 1 into two tiers separated by the shaded tasks. The shaded tasks are called *leaf inter-socket tasks*. Tasks above the leaf inter-socket tasks, including the leaf inter-socket tasks, are called *inter-socket tasks*, which belong to the inter-socket tier. Tasks in a subtree rooted with a leaf inter-socket task are called *intra-socket tasks*, which belong to the intra-socket tier. A subtree rooted with a leaf inter-socket task is called an *intra-socket subtree*. For example, in Fig. 1, tasks in an ellipse consist in an intra-socket subtree. The goal of CATS is to schedule tasks in the same intra-socket subtree within the same socket. In this way, CATS can ensure γ_1 and γ_2 (or γ_3 and γ_4) to be executed in the same socket.

However, to achieve the optimal scheduling, it is very challenging to find the proper leaf inter-socket tasks so that tasks in the same intra-socket subtree will be able to utilize the shared cache efficiently. If an intra-socket subtree is too large, the involved data can be too large to fit into the shared cache of the socket. On the other hand, if an intra-socket subtree is too small, the workload of the subtree can be too small to get better balanced among the cores of the same socket.

CATS uses an online DAG partitioner to find leaf inter-socket tasks and partition an execution DAG into two tiers. When CATS starts to execute a parallel program, the partitioner first profiles the program in the first iteration. Based on the profiling information, the online DAG partitioner adaptively divides the execution DAG into two tiers (to be discussed in Section 3.2). According to our first observation of parallel programs, the collected profiling information in the first iteration can be used to predict the execution behavior of the following iterations. Therefore, an optimal partitioning of DAG based on the profiling information of the first iteration will also be optimal for the following iterations.

After the runtime partitioning of the DAG, a bi-tier task-stealing algorithm is adopted in CATS to schedule tasks in the two tiers differently. The inter-socket tasks are scheduled across sockets, while the tasks in the same intra-socket subtree are scheduled within the same socket. CATS ensures that each socket can only execute one intra-socket subtree at the same time to avoid cache pollution. In this way, the shared data can be reused without reloading among tasks within an intra-socket subtree. That is, the scheduling in Fig. 2(a) can be enforced to reduce cache misses.

Fig. 3 illustrates the detailed processing flow of a parallel program in CATS.

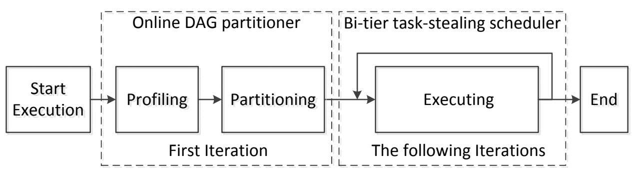


Figure 3. The processing flow of a parallel program in CATS.

3. Cache Aware Task-Stealing

This section presents CATS, a Cache Aware Task-Stealing scheduler. First, we give the CATS runtime environment. Then we describe an online DAG partitioner for dividing the execution DAG into two tiers. Third, we present the bi-tier task-stealing algorithm, the task-generating policy and the implementation details in CATS. Lastly, we discuss the time and space bounds of CATS.

3.1 CATS runtime environment

To support the processing flow in Fig. 3, we have built a runtime environment for CATS as follows. For an M-socket N-core architecture, CATS launches $M \times N$ workers (i.e., threads) at runtime and affiliates each worker with one individual hardware core as shown in Fig. 4. For convenience of presentation, we use the term *core* to mean a worker in the rest of the paper.

In each socket, only one core is selected as the head core of the socket to look after the inter-socket task scheduling. In our implementation, we choose “core 0” in each socket as the socket’s header core.

In order to schedule inter-socket tasks and intra-socket tasks in different ways in bi-tier task-stealing, CATS creates an *inter-socket task pool* for each socket to store inter-socket tasks, and an *intra-socket task pool* for each core to store intra-socket tasks, as shown in Fig. 4. A task pool is a double-ended queue (deque) that is used to store tasks.

During the first iteration of a parallel program, all the tasks are generated and pushed into intra-socket task pools when they are generated. In this case, tasks are scheduled adopting traditional task-stealing policy. That is, in the first iteration, tasks in intra-socket task pools can be scheduled across sockets since the profiling information has not been collected and thus the execution

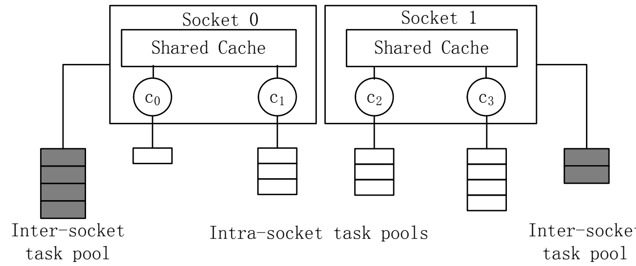


Figure 4. CATS runtime environment in a dual-socket dual-core architecture. Each socket has an inter-socket task pool and each core has an intra-socket task pool.

DAG has not been partitioned. In the following iterations, tasks are generated and pushed into different pools accordingly. If core c in socket ρ generates a task γ that is an inter-socket task, γ is pushed into ρ’s inter-task pool. Otherwise, if γ is an intra-socket task, it is pushed into c’s intra-socket task pool.

We present the online DAG partitioner and the bi-tier task-stealing scheduler in detail in the following sections.

3.2 Online DAG partitioner

As explained in Section 2, to partition an execution DAG into the inter-socket tier and the intra-socket tier optimally, the most challenging problem is to find the proper leaf inter-socket tasks. Once the proper leaf inter-socket tasks are identified, the DAG can be easily divided into two tiers: all the tasks above the leaf inter-socket tasks (including the leaf inter-socket tasks) belong to the inter-socket tier, and those tasks in the subtrees rooted with leaf inter-socket tasks belong to the intra-socket tier.

An optimal partitioning of an execution DAG should satisfy two constraints. The first constraint is that, for any intra-socket subtree ST, the involved data of all the tasks in ST is small enough to fit into the shared cache of a socket. The second constraint is that an intra-socket subtree ST should be large enough to allow a socket to have sufficient intra-socket tasks.

To fulfill the two constraints when dividing an execution DAG, for any task γ in the execution DAG, CATS should collects its involved data size. For convenience of description, we use *Size Of Involved Data* (SOID) to represent the involved data size of a task γ. That is, SOID includes the data accessed by all tasks in the subtree rooted with γ. Once the SOIDs for all tasks in the execution DAG are known, the online DAG partitioner can divide the execution DAG into two tiers optimally.

3.2.1 Online Profiling

In order to collect SOIDs of all the tasks in the execution DAG, CATS profiles the program during the first iteration of the execution. During the online profiling, we use the hardware Performance Monitoring Counters (PMC) [3] to collect cache misses, based on which the SOIDs for all tasks are calculated. The performance counter event we have used is the last level private data cache (e.g. L2 in AMD Quad-core Opteron 8380) misses. That is, we have used the performance counter event “07Eh” with mask of “02h” to collect the last level private data cache misses in AMD Quad-core Opteron 8380. For detailed information of the performance counter events, refer to *BIOS and Kernel Developer’s Guide* of the corresponding processor. Though it is straightforward to collect the event statistics of the last level private data cache misses in modern multi-core machines like X86_64, it is very tricky to calculate the SOIDs of the tasks based on the last level private data cache misses.

First, limited by the hardware PMCs, a core can only collect the cache misses of its own, but a task may have multiple child tasks executing on different cores. Therefore, it is impossible to collect the overall cache misses for a task directly.

Second, it is nontrivial to relate the private cache misses to the SOID of a task. For a task γ that runs on a core c in socket ρ, if γ fails to get its data from the last level private cache of c, it requests the data from the shared cache of ρ. Since c does not execute other tasks when it is executing γ, the last level private cache misses of c are totally caused by γ. The last level cache misses of c can be used to approximate to the size of data accessed by γ for the following reasons. Many memory-bound applications adopt data parallelism. As mentioned in our second observation in Section 2.2, only the leaf tasks physically access data. The data of leaf tasks do not have much overlapping with each other. Even when two neighbor leaf tasks have a small portion of shared data, the chances for them to be executed in the same core are small in a random task-stealing scheduler, which is adopted during the profiling stage. Therefore, the above approximation is accurate enough for us to calculate the SOIDs of all tasks.

Based on the collected last level private cache misses of γ, its SOID is calculated as follows. If γ is a leaf task, the number of cache misses of γ times the cache line size (e.g., 64 bytes in AMD Quad-core Opteron 8380) is γ's SOID. Otherwise, if γ is not a leaf task, its SOID is the sum of its cache misses times the cache line size plus the SOIDs of all its child tasks. Given a task β with n sub-tasks β_1, β_2, ..., β_n. Suppose M is β's number of cache misses times the size of cache line, and the SOIDs of its child tasks are S_1, S_2, ..., S_n respectively, then β's SOID, denoted by S_β, is calculated in Eq. 1.

$$S_\beta = M + \sum_{i=1}^{n} S_i \tag{1}$$

Based on Eq. 1, Fig. 5 presents an example of calculating SOIDs for all the tasks. In the figure, S_i is the SOID for leaf task γ_i, but represents the size of data physically accessed by the task itself for non-leaf tasks. In fact, for many memory-bound applications, S_i for non-leaf tasks is very small, if it is not zero, since non-leaf tasks do not physically access data.

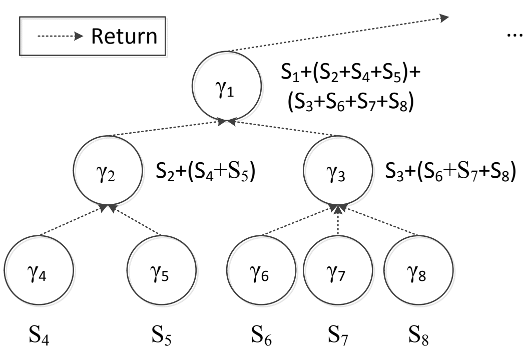


Figure 5. Collect Size Of Involved Data (SOID) for tasks.

As shown in Fig. 5, the SOID of a task is returned to its parent task when it is completed. For example, in Fig. 5, γ_2's SOID is added to γ_1's SOID when γ_1 is completed. Therefore, when all the tasks in the first iteration are completed, the SOIDs of all the tasks can be calculated.

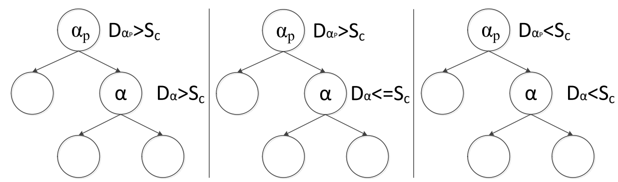
3.2.2 DAG Partitioning

Based on the SOIDs of tasks that are collected in the first iteration, the online DAG partitioner divides the execution DAG into inter-socket tier and intra-socket tier automatically.

To satisfy the aforementioned constraints, the online DAG partitioner identifies leaf inter-socket tasks as follows. For a task α and its parent task α_p, let D_α and D_{α_p} represent SOIDs of α and α_p respectively. α is a leaf inter-socket task if and only if D_α is smaller than the size of the shared cache and D_{α_p} is larger than the size of the shared cache.

More precisely, given a task α and its parent task α_p, our DAG partitioning method determines α's tier as follows.

- If both D_{α_p} and D_α are larger than the shared cache of a socket, α is an inter-socket task, as shown in Fig. 6(a).
- If D_{α_p} is larger than the shared cache and D_α is smaller than the shared cache of a socket, α is a leaf inter-socket task, as shown in Fig. 6(b).
- If both D_{α_p} and D_α are smaller than the shared cache, α is an intra-socket task, as shown in Fig. 6(c).



(a) α is an inter-socket task (b) α is a leaf inter-socket task (c) α is an intra-socket task

Figure 6. Conditions that α is an inter-socket task, leaf inter-socket task or intra-socket task.

After the profiling and the partitioning, the online DAG partitioner has already divided the execution DAG into two tiers optimally. Then, based on the optimal partitioning, bi-tier task-stealing can be adopted to schedule tasks for optimizing shared cache in the following iterations.

In order to identify the same task in the following iterations, during the execution of a parallel program, each task is given an identifier (string) according to the spawning relationship between tasks. If a task γ's identifier is S, then its ith sub-task's identifier is S_i. For example, Fig. 1 shows the way of constructing identifiers for tasks. The strings beside the tasks are the identifiers in Fig. 1. The identifiers of all the completed tasks are saved in a hash table with their SOIDs. When a new task is spawned, CATS tries to find its identifier in the hash table. If the identifier is found, it means the first iteration has completed since a new task in the same location of the execution DAG has been spawned. In this case, CATS uses the bi-tier task-stealing scheduler to schedule tasks based on their tiers which are decided according to their SOIDs as shown above.

Note that, in our implementation, we obtain the size of the shared cache from */proc/cpuinfo* by the CATS runtime system. In this way, all the needed information for optimal bi-tier task-stealing is obtained automatically by the runtime system of CATS. To this end, CATS can automatically improve the performance of parallel application without human intervention.

3.3 Bi-tier task-stealing scheduler

Task-stealing algorithm is used by a free core to obtain or steal a new task. When CATS starts to execute a parallel program, during the first iteration, CATS has not partitioned its execution DAG into two tiers. Therefore, the cores adopt the traditional task-stealing algorithm to obtain or steal a new task in the first iteration. In the following iterations, CATS adopts a bi-tier task-stealing algorithm to schedule tasks so that tasks in a subtree rooted with a leaf inter-socket task are scheduled to the same socket. Since traditional task-stealing has been discussed in detail in [7], this section only presents the bi-tier task-stealing in CATS.

When a core c in socket ρ is free, it first tries to obtain a task from its own intra-socket task pool. If its own task pool is empty, c tries to steal a task from the intra-socket task pools of other cores in ρ. If the task pools of all the cores in ρ are empty, the head core of ρ tries to obtain a task from its own inter-socket task pool. If its inter-socket task pool is empty, the head core tries to steal an inter-socket task from other sockets.

In CATS, only the head core of each socket can steal inter-socket tasks so that the lock contention of the inter-socket task pools is reduced. In addition, cores in the same socket are not allowed to execute tasks in different intra-socket subtrees at the same time. This policy can avoid the situation where different intra-socket subtrees pollute the shared caches with different data sets. The downside of the policy is that some cores in a socket may be idle waiting for other cores to finish their tasks. An alternative policy is to allow a socket to execute tasks from more than one intra-socket subtrees at the same time. This alternative policy can ensure most cores are busy, but different intra-socket subtrees may pollute the shared caches, which leads to more cache misses. For the memory-bound applications that CATS is targeting, the cache misses are more critical to the overall performance according to our experimental results. Therefore, we have adopted the first policy in CATS.

3.4 Task generating Policy

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 [19]. Parent-first policy works better when the steals are frequent and the execution DAG is shallow [19]. 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 [8]. Child-first policy works better when the steals are infrequent [19].

During the first iteration of a parallel programs, tasks have not been divided into inter-socket tasks and intra-socket tasks. For the convenience of collecting SOID, we choose to adopt the parent-first policy in the first iteration.

After the execution DAG has been divided into two tiers, CATS generates inter-socket tasks with the parent-first policy and generates intra-socket tasks with the child-first policy. CATS adopts the parent-first policy for generating inter-socket tasks so that leaf inter-socket tasks can be generated as soon as possible. The parent-first policy is more efficient in this case because inter-socket tasks take short time and thus are frequently stolen. On the other hand, CATS adopts the child-first policy to generate intra-socket tasks. The child-first policy works better in this case because the leaf tasks take longer time and thus the steals are infrequent. Also the child-first policy is more space efficient.

3.5 Implementation

We implement CATS on the basis of MIT Cilk. All the Cilk programs can run in CATS without any modifications. MIT Cilk is one of the earliest parallel programming environments that implement task-stealing [16]. It extends C with three keywords: *cilk*, *spawn* and *sync* to declare parallelism in the program. *cilk* identifies a procedure as a *Cilk procedure*, *spawn* is used to generate a child task, and *sync* waits for all the child tasks that are generated by the current task, to return. MIT Cilk consists of a compiler and a scheduler. Cilk compiler, named as *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.

We have modified *cilk2c* to support both the parent-first and child-first task-generating policy while the original MIT *cilk2c* only support the child-first policy. At each spawn, CATS finds out whether the spawn happens in the first iteration of the program. If it is in the first iteration, the to-be-spawned task is spawned with the parent-first policy. If it is not in the first iteration and the to-be-spawned task's SOID is smaller than the size of the shared cache, CATS spawns the task with the child-first policy and pushes the task into the intra-socket task pool of the current core. Otherwise, CATS spawns the task with the parent-first policy and pushes the task into the inter-socket task pool of the current socket.

Since CATS aims to reduce shared cache misses, CATS may not work very well for CPU-bound applications since the cache misses have neutral effect on their performance. On the contrary, CATS may adversely affect the performance of the CPU-bound applications. To avoid the problem, an interface could be provided for users so that they can tell CATS that whether the to-be-executed program is CPU-bound or not through command line. However, even if the users could not figure out if the program is memory-bound or CPU-bound, CATS has provided the following mechanism to identify whether it is CPU-bound based on the profiling information collected in the first iteration. Given an MSMC architecture with k levels of caches and the cache miss penalty (i.e. the delay) 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 γ is CPU-bound or memory-bound. If $CMPI_\gamma$ is smaller than a predefined threshold, γ is CPU-bound. If most tasks are CPU-bound, CATS treats the program as a CPU-bound program. In this case, CATS simply generates and schedules tasks of CPU-bound programs in the same way as the traditional task-stealing algorithm. Experiment results in Section 4.3 show that the extra overhead in CATS for CPU-bound programs is negligible.

3.6 Theoretical time and space bounds

Since CATS also uses bi-tier task-stealing algorithm to schedule tasks in a parallel program except the first iteration, the theoretical time and space bounds of CATS are same to the bounds in CAB [11] although CATS can handle more complex execution DAGs. As in [11], we model the execution of a parallel program as an execution DAG $\mathcal{G}$. Each node in $\mathcal{G}$ represents a unit task, and each edge represents a dependence between tasks. Here we only give the time and space bounds of CATS. Detailed deduction of the time and space bounds can be found in [11].

The time bound of CATS in an M-socket N-core architecture can be expressed as Eq. 2.

$$T_{M*N}(\mathcal{G}) = O(\frac{T_1(\mathcal{G}_{ir})}{M} + \frac{T_1(\mathcal{G}_{ia})}{M \times N} + T_\infty(\mathcal{G})) \quad (2)$$

In Eq. 2, $T_1(\mathcal{G}_{ir})$ means the number of nodes in the inter-socket tier in $\mathcal{G}$ and $T_1(\mathcal{G}_{ia})$ means the number of nodes in the intra-socket tier in $\mathcal{G}$. According to the equation, the inter-socket tier is executed by only M head cores. However, in memory-bound applications, only the leaf tasks in the DAG process input data, while the inter-socket tasks only divide the input data into smaller parts. Therefore, for memory-bound applications, the main part of the execution time is spent by the leaf tasks, i.e., the intra-socket tasks. Our experiments show that the execution time of the inter-socket tier is often less than 5% of the overall execution time. Therefore, the time bound of Eq. 2 is very close to the traditional task-stealing schedulers such as Cilk for memory-bound applications.

The space bound of CATS is shown in Eq. 3.

$$S_{M*N}(\mathcal{G}) \leq \max\{K \times S_1(\mathcal{G}), M \times N \times S_1(\mathcal{G})\} \qquad (3)$$

In the equation, we assume there are K leaf inter-socket tasks. Since K is not much larger than M, the space bound has the same *O-notation* as the traditional task-stealing schedulers.

4. Evaluation

We use a Dell 16-core computer that has four AMD Quad-core Opteron 8380 processors (codenamed "Shanghai") running at 2.5 GHz to evaluate the performance of CATS. Each Quad-core socket has a 512K private L2 cache for each core and a 6M L3 cache shared by all four cores. The computer has 16GB RAM and runs Linux 2.6.29.

Since CATS is proposed to reduce cache misses, we use memory-bound benchmarks to evaluate the performance of CATS. However, CPU-bound benchmarks are also used to measure the extra overhead of CATS compared with random task-stealing.

Table 1. Benchmarks used in the experiments

Name	Bound	Description
Mandelbrot	CPU	Calculate Mandelbrot Set
Queens(15)	CPU	N-queens problem
FFT	CPU	Fast Fourier Transform
GA	CPU	Island Model of Genetic Algorithm
Knapsack	CPU	0-1 knapsack problem
Heat	Memory	Five-point heat
Heat-ub	Memory	Five-point heat (unbalance)
SOR	Memory	Successive Over-Relaxation
SOR-ub	Memory	Successive Over-Relaxation (unbalance)
GE	Memory	Gaussian elimination
GE-ub	Memory	Gaussian elimination (unbalance)

To evaluate the performance of CATS in different scenarios, we use benchmarks that have both balanced and unbalanced execution DAGs in the experiments. Table 1 lists the used CPU-bound and memory-bound benchmarks. *Heat-ub*, *GE-ub* and *SOR-ub* implement the same algorithm as *Heat*, *GE* and *SOR* respectively, except their execution DAGs are unbalanced trees. For example, we implement *Heat-ub* in Algorithm 1. According to the algorithm, the branching degree of tasks created from cilk procedure *heat* is 2 while the branching degree of tasks created from cilk procedure *heat2* is 4. Obviously, *Heat-ub*'s DAG is an unbalanced tree. *GE-ub* and *SOR-ub* are implemented in the similar way.

Algorithm 1 The source code skeleton of *Heat-ub*

```
cilk void heat (int start, int end) {
   int mid = (start + end) / 2;
   spawn heat2 (start, mid);
   spawn heat (mid, end);
   sync; return;
}
cilk void heat2 (int start, int end) {
   int quad = (end - start) / 4;
   spawn heat (start, start + quad);
   spawn heat (start + quad, start + 2 * quad);
   spawn heat2 (start + 2 * quad, start + 3 * quad);
   spawn heat (start + 3 * quad, end);
   sync; return;
}
```

As mentioned before, CATS affiliates each worker with a hardware core. However, MIT Cilk does not affiliate workers with the cores. Since the affiliation can improve the cache performance of the execution, it is unfair to compare CATS with the original MIT Cilk. Therefore, we have modified the MIT Cilk to affiliate each worker with a hardware core (denoted as *Cilk-a* for short), in order to ensure fair comparison.

In all of our experiments, we compare the performance of CATS with Cilk-a. Cilk-a uses the pure child-first policy to spawn and schedule tasks, while CATS flexibly uses both the child-first and parent-first policies to achieve the best performance. We implement all the task-stealing schedulers based on MIT Cilk. The MIT Cilk programs run with Cilk-a and CATS without any modification.

All benchmarks are compiled with "cilkc -O2", which is based on gcc 4.4.3. Furthermore, for each test, every benchmark is run ten times. Since the execution time is very stable, the average execution time is used in the final results.

4.1 Performance of memory-bound applications

Fig. 7 shows the performance of memory-bound benchmarks in Cilk-a and CATS with a 1024×512 matrix as the input data. For *GE* and *GE-ub*, the used input data is a 1024×1024 matrix. As we can see from Fig. 7, CATS can significantly improve the performance of memory-bound applications compared to Cilk-a while the performance improvement ranges from 35.3% to 74.4%.

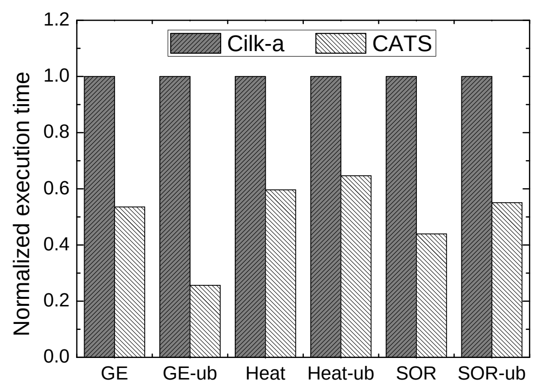


Figure 7. The performance of memory-bound benchmarks in Cilk-a and CATS.

To explain why CATS can improve the performance of memory-bound applications compared with Cilk-a, we collect the cache misses of all the benchmarks and list them in Table 2. Observed from the table, we can find that the shared cache (L3) misses are prominently reduced while the private cache (L1 and L2) misses are also slightly reduced in CATS compared with Cilk-a. Since CATS schedules tasks with shared data into the same socket, the shared cache misses have been significantly reduced.

Although scheduling tasks with shared data to the same socket only reduces the shared L3 cache misses, the affiliation of an intra-socket subtree with a socket in CATS can help reduce the L2 cache misses slightly. In CATS, for a task γ_i in an intra-socket subtree, if it is executed by core c in socket ρ, its neighbor tasks (i.e., γ_{i-1} and γ_{i+1}) are also executed by w as well unless they are stolen by other cores in ρ. In random task-stealing, however, any free cores can steal γ_i's neighbor tasks. There are fewer cores that can steal γ_i's neighbor tasks in CATS compared to random task-stealing. Therefore, the probability that neighbor tasks are executed by the same core is larger in CATS. For this reason, the private cache (e.g., L2) misses have also been slightly reduced in CATS.

Fig. 8 shows the SOIDs of *Heat* with a 1024×512 matrix as input data that are calculated as Eq. 1. The real involved data size of tasks in Fig. 8 are calculated as follows. Since *Heat* uses

Table 2. Cache misses in Cilk-a and CATS (*1E6)

Application	Scheduler	L1	L2	L3
GE	Cilk-a	60.8	58.8	14.5
	CATS	53.9	50.3	2.94
GE-ub	Cilk-a	37.2	37.1	10.7
	CATS	23.9	20	2.15
Heat	Cilk-a	82.7	79.6	24.8
	CATS	71.1	67.5	5.9
Heat-ub	Cilk-a	82.2	78.7	29.7
	CATS	71.3	67.6	3.72
SOR	Cilk-a	88.5	85	29.6
	CATS	70.7	66.2	4.75
SOR-ub	Cilk-a	89.8	85.5	30.7
	CATS	73.6	67.4	8.27

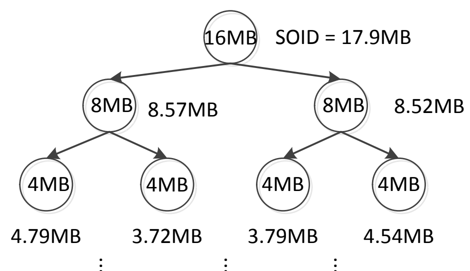


Figure 8. Calculated SOIDs of tasks in *Heat* with a 1024×512 matrix as input data.

two matrices of "double" during the execution, the overall effective input data size is $1024 \times 512 \times 16 \times 2 = 16$MB. Then, the data set is divided into two parts recursively. From the figure, we can find that the calculated SOIDs are not far away from the real involved data size. Therefore, our online DAG partitioner is reasonably effective.

4.2 Scalability of CATS

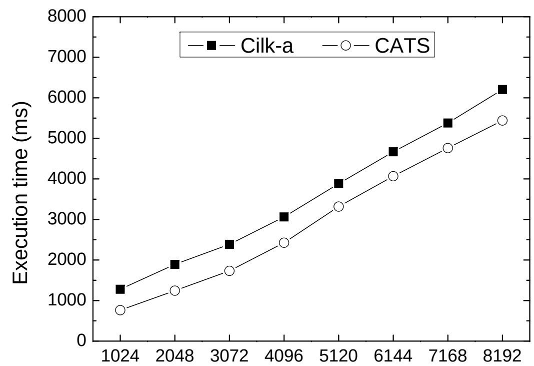
To evaluate scalability of CATS in different scenarios, we use benchmarks that have both balanced and unbalanced execution DAGs. In this experiment, we execute benchmarks with different input data sizes in CATS and Cilk-a. Their performance is then compared.

During the execution of all the benchmarks, every task divides its data set into several parts by rows to generate child tasks unless the task meets the cutoff point (i.e., the data set size of a leaf task). Since the data set size of the leaf tasks affects the measurement of scalability, we should ensure that the data set size of the leaf tasks is constant in our experiment. To satisfy this requirement, we use a constant cutoff point, 8 rows, for the leaf tasks, and a constant number of columns, 512, for the input data. We only adjust the number of rows of the input matrix in the experiment. In this way, we can measure the scalability of CATS without the impact of the granularity of the leaf tasks. In all the following figures, the x-axis represents the row number of the input data set.

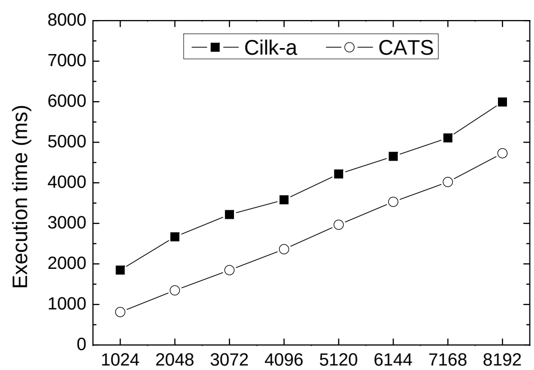
4.2.1 Balanced execution DAGs

We use *Heat* and *SOR* as benchmarks to evaluate the scalability of CATS in scenario that applications with balanced execution DAGs. Other benchmarks, such as *GE*, have similar results.

Fig. 9 shows the performance of *Heat* and *SOR* with different input data sizes in Cilk-a and CATS. From Fig. 9, we can see that *Heat* and *SOR* achieve better performance in CATS for all sizes of the input data up to 8192 rows compared with Cilk-a. When the input data size is small (i.e., 1024×512), CATS reduces 40.4% execution time of *Heat* and reduces 56.1% execution time of *SOR*.



(a) Performance of Heat



(b) Performance of SOR

Figure 9. Performance of Heat and SOR with different input data sizes.

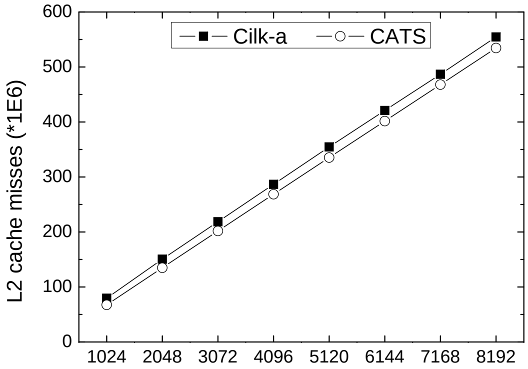
When the input data size is large (i.e., 8192×512), CATS reduces 12.3% execution time of *Heat* and reduces 21.1% execution time of *SOR*.

Fig. 10 shows the L2 and L3 cache misses of *Heat* with different input data sizes in Cilk-a and CATS. Observed from the figure, we can find that both the shared cache misses and the private cache misses are reduced in CATS compared with Cilk-a. The better performance of *Heat* in CATS results from the less cache misses in CATS compared with Cilk-a. When the input data size is small (1024×512), CATS can reduce 76.1% L3 cache misses and 15.2% L2 cache misses compared with Cilk-a. When the input data size is large (8192×512), CATS can reduce 55.9% L3 cache misses and 3.6% L2 cache misses compared with Cilk-a. Therefore, when CATS schedules regular applications with balanced execution DAGS, it is scalable. Other benchmarks show similar results of cache misses. We omit them here due to limited space.

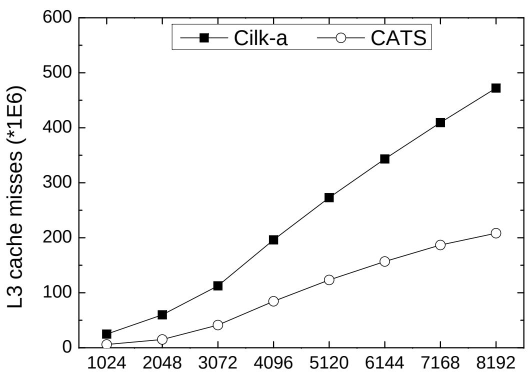
4.2.2 Unbalanced execution DAGs

We use *Heat-ub* and *SOR-ub* as benchmarks to evaluate the scalability of CATS in scenario that applications with unbalanced execution DAGs. Other benchmarks, such as *GE-ub*, have similar results.

Fig. 11 shows the performance of *Heat-ub* and *SOR-ub* with different input data sizes in Cilk and CATS. From Fig. 11 we can find that *Heat-ub* and *SOR-ub* also achieve better performance in CATS for all input data sizes compared with Cilk-a. When the input data size is small (i.e., 1024×512), CATS reduces 35.3% execution time of *Heat-ub* and reduces 44.9% execution time of *SOR-ub*. When the input data size is large (i.e., 8192×512), CATS reduces 11.4% execution time of *Heat-ub* and reduces 18% execution time of *SOR-ub*.

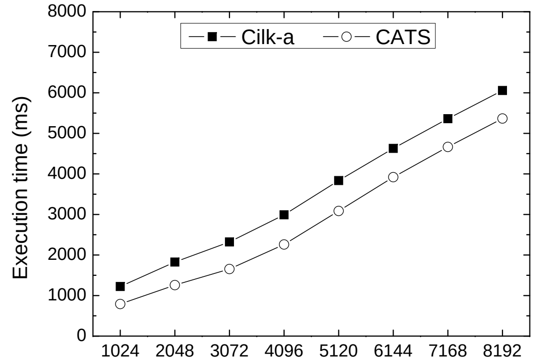


(a) L2 cache misses of Heat

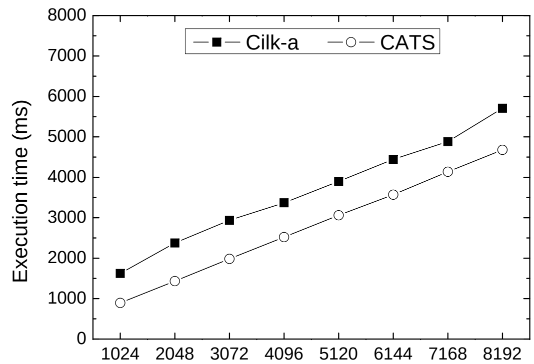


(b) L3 cache misses of Heat

Figure 10. L2 and L3 cache misses of Heat with different input data sizes.



(a) Performance of Heat-ub



(b) Performance of SOR-ub

Figure 11. Performance of Heat-ub and SOR-ub with different input data sizes.

Fig. 12 shows the L2 and L3 cache misses of *SOR-ub* with different input data sizes. Observed from the figure, we can find that both the shared cache misses and the private cache misses of *SOR-ub* are reduced in CATS compared with Cilk-a. The better performance of *SOR-ub* in CATS results from the less cache misses in CATS compared with Cilk-a. When the input data size is small, CATS can reduce 73.1% L3 cache misses and 21.2% L2 cache misses compared with Cilk. When the input data size is large, CATS can reduce 38.2% L3 cache misses and 5.2% L2 cache misses compared with Cilk. Other benchmarks show similar results of cache misses. We omit them here due to limited space.

As illustrated in Fig. 9 and Fig. 11, the execution time of benchmarks in both Cilk-a and CATS increases linearly in nearly the same speed with the increasing of input data size. Therefore, for all the input data size, CATS can reduce the execution time of memory-bound applications. Thus CATS is scalable in scheduling both balanced execution DAGs and unbalanced execution DAGs.

In addition, Fig. 10 and Fig. 12 further verify that CATS can also slightly reduce private cache misses by scheduling tasks with shared data into the same socket although it is proposed to reduce shared cache misses.

4.3 Performance of CPU-bound applications

Since CATS is proposed to reduce shared cache misses of memory-bound applications, it is neutral to CPU-bound applications. Therefore, for CPU-bound applications, CATS uses child-first policy to schedule the tasks as Cilk-a.

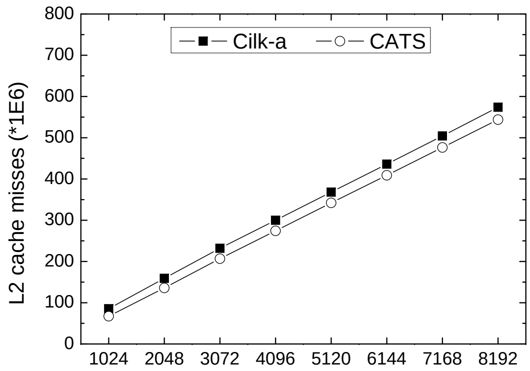
Fig. 13 shows the performance of CPU-bound benchmarks listed in Table 1 in Cilk-a and CATS. By comparing the performance of CATS with Cilk-a, we can find the extra overhead of CATS since they adopt the same policy to schedule CPU-bound applications. Observed from Fig. 13, we see the extra overhead of CATS is negligible compared with Cilk-a. The extra overhead of CATS mainly comes from the profiling overhead in the first iteration during the execution of a parallel program, when CATS can decide if the program is CPU-bound or memory-bound.

4.4 Discussion

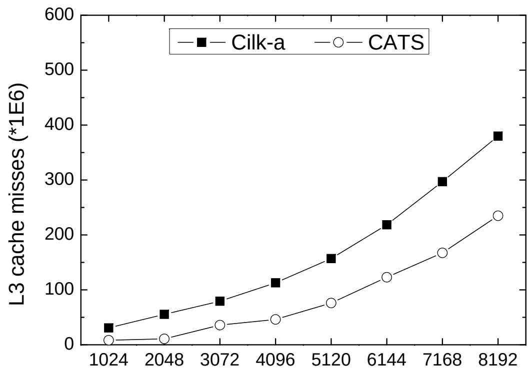
As mentioned before, CATS targets memory-bound programs whose execution DAGs are tree-shaped. Therefore the most important limitation of CATS is that CATS is not suitable for programs whose DAGs are not tree-shaped since the DAG partitioner is not applicable to non-tree DAGs. CATS is applicable to divide-and-conquer programs because they have tree DAGs. We have modified *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 programs that do not follow the divide-and-conquer pattern, CATS can simply use random task-stealing for task scheduling in the execution. Therefore, the above limitation does not affect the applicability of CATS since the compiler can identify the class of programs that are suitable for CATS.

5. Related Work

Reducing cache misses of parallel programs in parallel architectures is a popular research issue. However, many of the existing



(a) L2 cache misses of SOR-ub



(b) L3 cache misses of SOR-ub

Figure 12. L2 and L3 cache misses of SOR-ub with different input data sizes.

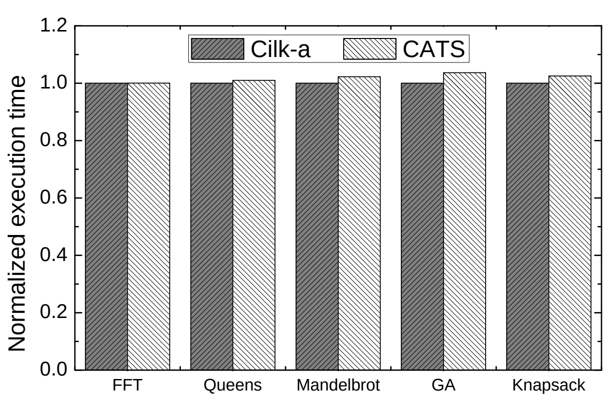


Figure 13. Performance of CPU-bound applications in Cilk-a and CATS.

works either need extra user-provided information or are not general enough for MSMC architectures.

In [27], MTS (Multi-Threaded Shepherds) was proposed to reduce cache misses in MSMC architecture. In MTS, when all the cores in a socket are free, the head core of the socket steals a batch of tasks from other sockets. However, MTS cannot ensure tasks executed by cores in the same socket have shared data, and thus cannot reduce shared cache misses in MSMC. In [4], CONTROLLED-PDF was proposed to reduce cache misses in single-socket multi-core architecture. The scheduler divided nodes of a DAG into *L2-supernodes* that contain data fit for the shared L2 cache. By executing L2-supernodes sequentially, the cache misses can be reduced. The scheduler needed users to provide space complexity function of the executed program and was only applicable to single-socket multi-core architecture. Also the paper did not evaluate the proposed scheduler through experiment. In [33], a less reused cache filter was proposed to filter out the less reused data so that the frequently reused data can stay in the cache.

Based on page-coloring technique, many works enable programmers to manage shared cache explicitly. In [28], a cache partitioning method was proposed. Based on the method, a cache control tool is implemented so that users can control the partitioning of cache. In [14], ULCC was proposed to explicitly manage and optimize last level cache usage by allocating proper cache space for different data sets of different threads. Although programmers may improve their programs by managing last level cache , the management is burdensome for programmers. In contrast, CATS can improve the last level cache (L3) performance of memory-bound applications automatically without extra user-provided information.

Cache-oblivious algorithms, which can achieve good cache performance by tuning the parallel programs carefully [6, 15, 31], were used in a *parallel cache-oblivious* (PCO) model [5]. Based on the PCO model, the authors described a scheduler to balance the cost of the cache misses across the processors. However parallel programs need to satisfy many restrictions so that the scheduler can perform efficiently. Especially, the paper did not really implement the PCO model and did not evaluate the model through experiment.

Task-stealing is popular for automatic load balancing inside parallel applications due to its high performance. Many works have been done on its adaption and improvement [10, 20–22, 26, 32].

There are also some works aiming to reduce cache misses in task-stealing on parallel architectures. In [1], a theoretical bound on the number of cache misses for random task-stealing was presented and a locality-guided task-stealing algorithm was implemented on a single-socket SMP. In [13], the authors analyzed the cache misses of algorithms using random task-stealing, focusing on the effects of false sharing. In [12], cache behaviors of task-stealing and a parallel depth-first scheduler were compared and analyzed on a multi-core simulator that has shared L2 caches among cores. It was proposed to promote constructive cache sharing through controlling task granularity. However, the above studies did not take the MSMC architecture into consideration, and thus did not target the reduction of shared cache misses as CATS does.

In [29], PWS (Probability Work-Stealing) and HWS (Hierarchical Work-Stealing) were proposed to reduce communications among different computers for hierarchical distributed platform. In PWS, processors had higher probability to steal tasks from processors in the same computer. HWS used a rigid boundary level to divide tasks into global tasks and local tasks which are similar to inter-socket tasks and intra-socket tasks in CATS. However, the boundary level in HWS must be given by users manually. Apart from their difference from CATS, it is also worth noting that PWS and HWS were proposed for reduction of communications in distributed environments.

In [11], a task-stealing scheduler, called CAB, is proposed to reduce shared cache misses in MSMC. Similar to HWS, CAB used a rigid boundary level to divide tasks into global tasks and local tasks. Though the boundary level is calculated at run-time, users have to provide a number of command line arguments for the scheduler to calculate the boundary level. If the arguments are not correct, the performance of applications may degrade seriously. In addition, CAB is not as adaptive as CATS since it cannot work with irregular and unbalanced execution DAGs that CATS works with.

6. Conclusions

Traditional task-stealing schedules tasks randomly to different cores. Although the random scheduling works efficiently in multi-core processor, it tends to pollute shared cache in MSMC architectures. To solve the problem, we have designed and implemented the CATS scheduler that requires no extra user information. Based on features of parallel programs, CATS uses an online DAG partitioner to divide execution DAG into inter-socket tier and intra-socket tier based on profiling information that is collected during the first iteration of the program. Furthermore, by scheduling tasks from an intra-socket subtree within the same socket, the shared cache misses are reduced significantly. Experimental results demonstrate that CATS can achieve up to 74.4% performance gain for memory-bound applications compared with random task-stealing and the extra overhead of CATS for CPU-bound applications is negligible.

One potential avenue of future work is to explore task-stealing in asymmetric architectures and design a special task-stealing scheduler to schedule tasks with different features onto different cores optimally in order to better utilize the system resources. Another promising future research direction is to optimize instruction caches by schedule tasks that execute the same instructions onto the same core.

References

[1] U. Acar, G. Blelloch, and R. Blumofe. The data locality of work stealing. *Theory of Computing Systems*, 35(3):321–347, 2002.

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

[3] R. Azimi, M. Stumm, and R. Wisniewski. Online performance analysis by statistical sampling of microprocessor performance counters. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 101–110. ACM, 2005.

[4] G. Blelloch, R. Chowdhury, P. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 501–510. Society for Industrial and Applied Mathematics, 2008.

[5] G. Blelloch, J. Fineman, P. Gibbons, and H. V. Simhadri. Scheduling irregular parallel computations on hierarchical caches. In *Proceedings of the 20th ACM Symposium on Parallel Algorithms and Architectures*, San Jose, California, June 2011.

[6] G. Blelloch, P. Gibbons, and H. Simhadri. Low depth cache-oblivious algorithms. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 189–199. ACM, 2010.

[7] R. D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Sept. 1995. MIT Laboratory for Computer Science Technical Report MIT/LCS/TR-677.

[8] 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*, 37(1):55–69, Aug. 1996.

[9] D. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1997.

[10] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, page 28. ACM, 2005.

[11] 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*, Taipei, Taiwan, 2011. IEEE.

[12] S. Chen, P. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. Mowry, et al. Scheduling threads for constructive cache sharing on CMPs. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, page 115. ACM, 2007.

[13] R. Cole and V. Ramachandran. Analysis of Randomized Work Stealing with False Sharing. *ArXiv e-prints*, Mar. 2011.

[14] X. Ding, K. Wang, and X. Zhang. ULCC: a user-level facility for optimizing shared cache performance on multicores. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 103–112, 2011.

[15] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297, New York, Oct. 17–19 1999.

[16] 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*, pages 212–223, Montreal, Canada, June 1998. ACM.

[17] A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16(4):276–291, 1992.

[18] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message passing interface*. MIT Press, 1999.

[19] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *IPDPS'09: Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12. IEEE Computer Society, 2009.

[20] Y. Guo, J. Zhao, V. Cave, and V. Sarkar. Slaw: a scalable locality-aware adaptive work–stealing scheduler. In *the 24th IEEE International Parallel and Distributed Processing Symposium*, 2010.

[21] D. Hendler, Y. Lev, M. Moir, and N. Shavit. A dynamic-sized non-blocking work stealing deque. *Sun Microsystems, Inc. Technical Reports; Vol. SERIES13103*, page 69, 2005.

[22] D. Hendler and N. Shavit. Non-blocking steal-half work queues. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, page 289. ACM, 2002.

[23] D. Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43. ACM, 2000.

[24] 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 computing*, pages 25–36. ACM, 2010.

[25] C. Leiserson. The Cilk++ concurrency platform. In *Proceedings of the 46th Annual Design Automation Conference*, pages 522–527. ACM, 2009.

[26] M. M. Michael, M. T. Vechev, and V. A. Saraswat. Idempotent work stealing. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 45–54. ACM, 2009.

[27] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, and J. F. Prins. Scheduling task parallelism on multi-socket multicore systems. In *Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '11, pages 49–56. ACM, 2011.

[28] S. Perarnau, M. Tchiboukdjian, and G. Huard. Controlling cache utilization of hpc applications. In *Proceedings of the international conference on Supercomputing*, pages 295–304. ACM, 2011.

[29] J.-N. Quintin and F. Wagner. Hierarchical work-stealing. In *Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, EuroPar'10, pages 217–229. Springer-Verlag, 2010.

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

[31] R. Strzodka, M. Shaheen, D. Pajak, and H.-P. Seidel. Cache oblivious parallelograms in iterative stencil computations. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 49–59, 2010.

[32] L. Wang, H. Cui, Y. Duan, F. Lu, X. Feng, and P. Yew. An adaptive task creation strategy for work-stealing scheduling. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 266–277. ACM, 2010.

[33] L. Xiang, T. Chen, Q. Shi, and W. Hu. Less reused filter: improving l2 cache performance via filtering less reused lines. In *Proceedings of the 23rd international conference on Supercomputing*, pages 68–79. ACM, 2009.

[34] J. Zhang, Z. Huang, W. Chen, Q. Huang, and W. Zheng. Maotai: View-Oriented Parallel Programming on CMT processors. In *37th International Conference on Parallel Processing*, pages 636–643. IEEE, 2008.