Adaptive Scheduling of Multiprogrammed Dynamic-Multithreading Applications

Zhe Wang, Chen Xu, Kunal Agrawal

Washington University in St. Louis

Jing Li

New Jersey Institute of Technology

Abstract

Modern parallel platforms, such as clouds or servers, are often shared among many different jobs. However, existing parallel programming runtime systems are designed and optimized for running a single parallel job, so it is generally hard to directly use them to schedule multiple parallel jobs without incurring high overhead and inefficiency. In this work, we develop AMCilk (Adaptive Multiprogrammed Cilk), a novel runtime system framework, designed to support multiprogrammed parallel workloads. AMCilk has client-server architecture where users can dynamically submit parallel jobs to the system. AMCilk has a single runtime system that runs these jobs while dynamically reallocating cores, last-level cache, and memory bandwidth among these jobs according to the scheduling policy. AMCilk exposes the interface to the system designer, which allows the designer to easily build different scheduling policies meeting the requirements of various application scenarios and performance metrics, while AMCilk transparently (to designers) enforces the scheduling policy. AMCilk also enables its use in cloud environment where other processes may be sharing the system with AMCilk. In this scenario, an external scheduler can change the resource availability for AMCilk and AMCilk seamlessly adapts to these changes. The primary feature of AMCilk is the low-overhead and responsive preemption

Email addresses: `{zhe.wang, chenxu, kunal}@wustl.edu` (Zhe Wang, Chen Xu, Kunal Agrawal), `jingli@njit.edu` (Jing Li)

mechanism that allows fast reallocation of cores between jobs. Our empirical evaluation indicates that AMCilk incurs small overheads and provides significant benefits on application-specific criteria for a set of 4 practical applications due to its fast and low-overhead core reallocation mechanism.

Keywords: multiprogrammed, parallel computing, Cilk, cloud

2010 MSC: 00-01, 99-00

---

1. Introduction

In recent years, the number of cores on multiprocessor and multicore systems has been increasing at a rapid rate. With this trend, there is an increasing interest in running many parallel jobs on a single machine at the same time, especially in the context of shared environments such as clouds and shared clusters. However, most parallel runtime systems, such as Cilk variants [1, 2, 3, 4], OpenMP [5], and TBB [6], are designed to run a single parallel job. To run multiprogrammed workloads, one must frequently instantiate one runtime system for each job. Since these runtime systems are unaware of being in a multiprogrammed environment and often assume that they have a certain number of cores, say p (often the entire machine), dedicated to running their single job, they create p pthreads, pin them to each of these cores and use them to execute for the duration of the job. This leads to suboptimal performance for jobs in these environments.

For multiprogrammed environments, the system scheduler must decide how to allocate system resources among the different jobs in the system. This allocation depends on the performance goal of the system and different applications with multiprogrammed workloads may have different performance goals. For instance, an interactive web service running on a cloud may care about minimizing some function of the latency of the jobs. On the other hand, a real-time application running on an embedded device may require that jobs meet their deadlines. There has been significant theoretical research on designing schedulers for various performance goals, e.g., minimizing some function of the job latencies [7, 8, 9, 10, 11, 12, 13, 14, 15, 16] and guaranteeing no deadline misses [17, 18, 19, 20, 21, 22]. However, most of these schedulers have either not been implemented or implemented using a custom-built system for that application scenario.

In this work, our goal is to design a high-performance, flexible and extensible framework for enabling multiprogrammed workloads in a shared environment. Since the different multiprogrammed parallel workloads have various job arrival patterns, job memory access characteristics, requirements and performance objectives, we want to design the parallel runtime system that enables the following functionalities: (1) ***Online arrival:*** Jobs can arrive online, and the scheduler does not need to know what jobs will arrive in the future; (2) ***Dynamic reallocation:*** The scheduler can dynamically increase or decrease the number of cores allocated to a job while the job is executing; (3) ***Efficient execution:*** The job must efficiently use the cores that are assigned to it at any moment using an efficient parallel scheduling algorithm such as work-stealing [1]; (4) ***Cache management:*** The job scheduler can support cache partitioning and memory bandwidth allocation, as a complement to core allocations, to mitigate the cache and memory bandwidth contention and support quality of service; and (5) ***External Resource Control:*** When AMCilk is sharing a machine with other processes (say in a shared cloud environment) an external scheduler should be able to control the resource occupancy by the AMCilk runtime.

In most parallel runtime systems, dynamically changing the number of cores allocated to the job is difficult and expensive for multiple reasons. Since multiprogrammed systems often run each job in its process, deallocating a core from one job and allocating it to another often involves an operating system (OS) call. Since the OS may not be aware of what is happening within the job, the thread running on a deallocated core may be holding a lock or be in some unsafe state when it is de-scheduled, compromising the efficiency of the parallel program. Moreover, the kernel operations involved when reallocating cores are likely to be expensive. Finally, the job scheduler may have high inter-process communication overhead for collecting runtime information required to make scheduling decisions.

In this paper, we take a different approach. We design AMCilk (adaptive multiprogrammed Cilk), a parallel runtime framework extending the Cilk runtime systems to efficiently support multiprogrammed scenarios in a shared environment. Specifically, AMCilk has the following features:

- AMCilk allows a system administrator to implement their preferred scheduling policy to allocate cores among different jobs to optimize the application-specific performance criterion by exposing an easy interface. The AMCilk framework then transparently (to the system administrator) implements this policy by automatically reallocating cores as dictated by the policy.

- AMCilk's client-server architecture allows jobs to be submitted online, start new jobs dynamically and return results of completed jobs to clients.

- AMCilk concurrently runs multiple parallel jobs in a single runtime system, so that the AMCilk scheduler can access the full runtime information of jobs and enforce core reallocation with low overhead.

- AMCilk develops a safe, low-cost, and responsive preemption mechanism, which allows reallocating cores between jobs in microseconds while the jobs are running. Thus, it has little performance penalty on the jobs. Note that the "preemption" in this paper denotes the action of stopping the execution of a parallel job on a processor, and the AMCilk runtime system enables this preemption mechanism.

- AMCilk exposes interfaces that use the hardware-level cache partitioning and memory bandwidth allocation to restrict the interference between jobs and to control the quality of service when multiprogrammed jobs compete for the last-level cache and memory bandwidth.

- AMCilk can be run in an environment where other processes share the hardware resources with it. To enable this, AMCilk exposes its resource allocation interface to external schedulers which can then control the resource occupancy of AMCilk. AMCilk then seamlessly adapts to this new resource allocation.

- In order to enable an intelligent allocation by the external scheduler, AMCilk exposes its runtime information to these schedulers using a subscription model.

Therefore, when building applications using AMCilk, system administrators can customize the core allocation, cache partitioning and memory bandwidth allocation policy via AMCilk interfaces, without needing to understand the implementation details.

Our evaluation indicates that the overheads of starting a new job, completing a job, reallocating cores, etc., within AMCilk are small, and the core reallocation adds a minimal performance penalty on job executions. Moreover, we implemented application scenarios using AMCilk to understand whether AMCilk provides performance improvement to their application-specific criteria. In particular, we developed four applications by implementing their scheduling algorithms via the AMCilk policy-customization interface. The first one [10] has the goal of minimizing the average latency of online parallel jobs, such as those in interactive services. We find that the implementation based on AMCilk provides a performance advantage of between 60 to 70% over the previous implementation (which was used in the experiment of [10]), which uses the same scheduling policy — the performance improvement is purely due to AMCilk's ability to reallocate cores faster than the previous implementation. The second one is an elastic real-time application [22] with periodic tasks that must meet deadlines, where some tasks can vary their demand causing other tasks to adjust their deadlines accordingly. Again, we see that AMCilk provides better responsiveness to the demand change, providing better performance to the application. The third application is an application that dynamically adapts the number of cores according to the parallelism of the applications and requires that we monitor the jobs to adjust the core allocation. We see that the

AMCilk implementation successfully adapts to the changing parallelism providing better performance than the best static allocation. The forth application demonstrates the importance of cache and memory bandwidth partitioning in multiprogrammed environments. In addition, the final experiment shows the efficiency of the subscription functionality for external schedulers.

This paper extends from [23] which has been published on HiPC 2020. In the HiPC version, we develop AMCilk, as a framework for scheduling multiprogrammed parallel workloads. In this paper, we extend AMCilk from two aspects: (1) In the HiPC version, we assume that the physical machine dedicates to the AMCilk runtime system. However, in the cloud environment, AMCilk may co-run with other applications. It would be greatly beneficial if AMCilk could expose interfaces and allow external schedulers to dynamically control the resource occupancy of AMCilk. Thus, this paper presents the support of the cloud environment (Section 4). (2) In the HiPC version, we introduce the AMCilk policy-customization interface which allows system designers to customize the scheduling policy of AMCilk. However, there lacks a demo to show how to use the interfaces to prototype a scheduler. In this paper, we present a concrete example of developing schedulers by using the AMCilk policy-customization interface (Section 3.2).

2. Background

AMCilk is implemented for the Cilk language using a home-grown Cheetah runtime system, which is similar to Intel's Cilk Plus runtime system [4]. Cilk [2] is a parallel programming language that extends C, while Cilk Plus was designed later for C++. Here we describe the key features of Cheetah that are critical for understanding the design of AMCilk.

Cilk Plus language and Cheetah runtime system. Cilk Plus extends C++ with additional keywords, principally including ***spawn*** and ***sync***. A function that is ***spawned*** may execute in parallel with the continuation of its parent function. The ***sync*** keyword indicates that all function instances spawned by the current function must return before the next instruction. Therefore, the programmer expresses the *logical parallelism* of the program, while the Cheetah runtime system is responsible for scheduling this program on the given number of cores. The compiler and linker compile the program by inserting calls to the runtime system at function spawn, return, and sync. The program's main function is compiled as the `cilk_main` function, while the newly added `main` function performs runtime initialization by creating p threads, one for each core, and pins them on their cores. It also sets up data structures for scheduling this program on these threads. One key data structure is a ***worker*** for each thread, which keeps track of information about that thread from the perspective of the program — for most of this paper, we will use the term worker and thread interchangeably. After initialization, the runtime calls the `cilk_main` function to begin executing the program.

Work-Stealing. Work-stealing [1] is a theoretically good and practically efficient scheduling algorithm used by many programming languages and libraries, such as Cilk variants [1, 2, 3], OpenMP [5], and Intel's TBB [6]. Same as common Cilk variants, in the Cheetah runtime system, each worker maintains its own deque (a double-ended queue) of stack frames and pushes/pops stack frames from the bottom of the deque. If a worker's deque is empty, it becomes a ***thief***, picks a random victim among the other workers, and steals the frame from the top of the victim's deque and starts executing it.

THE Protocol. A worker pushes and pops frames from the bottom of its own deque, while a thief might steal work from the top of another worker's deque. Therefore, if there is only one frame on a deque, any thief who tries to steal it must synchronize with the owner to ensure consistency. The Cheetah runtime system employs the ***THE*** protocol [3] to perform the synchronization efficiently. The THE protocol uses three shared atomic variables: T, H, and E. T and H mark the head and tail of the deque, and E is an exception pointer and marks a place where T cannot cross over.

Generally, E and H both point at the head of a deque, while T points at the tail. When a worker pushes a frame on the deque, it simply increments T. When a thief tries to steal from the top of the deque, it grabs the lock of the victim's deque and increments E. If E $\leq$ T, the thief steals the top frame and increments H; otherwise, it gives up and restores E. It then releases the deque lock. When a worker tries to pop a frame, it decrements T and then compares it with E. If E$\leq$T, then the worker can pop without getting any locks. If E $>$ T, the worker calls an exception handler within the runtime system. Generally, this means that some thief is trying to steal while the victim is trying to pop. In this case, the victim also tries to get the deque lock, and either the thief or the victim wins based on who gets the lock.

This E pointer can also be used to trigger exceptions of other kinds — essentially, by setting E to be larger than T, we can force the thread to enter the exception handling routine within the runtime system and then modify the exception handling routine to perform other operations. We will use this functionality in AMCilk to inform the worker to perform core reallocations — described in Section 3.

3. AMCilk Scheduling Framework

This section describes the key implementation details of the AMCilk scheduling framework. In particular, AMCilk uses a ***client-server architecture*** (§3.1) to support online arrival and completion of jobs. This design also separates the responsibility between the system administrator and users. The users simply submit their jobs to a server while the server runs the parallel jobs concurrently in a runtime system. The scheduling policy of the server is managed by the system administrator.

The AMCilk scheduling framework provides ***policy-customization interfaces*** (§3.2) that allow system administrators to easily and flexibly customize the scheduling policy that allocates shared resources, including cores, last-level cache, and memory bandwidth, to concurrent parallel jobs. In particular, AMCilk provides an integrated and easy-to-use interface that implements a ***de-***

centralized AMCilk scheduler (§3.3), which is called automatically at predefined events such as job arrivals, job completions, and timer interrupts.

The scheduler may change the allocation between jobs while the jobs are running — to support this, we implemented a ***responsive and low-cost core reallocation mechanism*** (§3.4). This preemption mechanism makes a good trade-off between the system overheads, responsiveness to the scheduling decisions, and transparency to user programs, by leveraging the exception mechanism in the Cilk runtime.

Finally, to retain the theoretical guarantees of work-stealing for a parallel job, the AMCilk scheduling framework augments work-stealing within each job on the assigned cores with an ***efficient work resumption mechanism*** (§3.5). It ensures that when a core is taken away from a job (decided by the scheduling policy and enforced by the preemption mechanism), the leftover work of this job on the core gets completed in a timely manner by other cores allocated to this job.

3.1. Client-Server Architecture

Figure 1 illustrates the conceptual client-server architecture of AMCilk. A client (i.e., user) creates a ***job request*** struct, which stores the program id (indicating which program to run) and its input parameters. It submits the job request to the server via a pipe. AMCilk has a dedicated ***request receiver*** thread (pinned to a dedicated core) that listens for requests and on receiving a request, pushes it into a FIFO ***job request buffer***. The AMCilk scheduler takes job requests from the head of the buffer, parses the request, and prepares to run the executable of the corresponding program. When a job finishes, the server sends the result to the client. The result is the return value or the location where the return value stored. Both request receiver and AMCilk scheduler are nonblocking — they do not wait for a job request to complete before starting on the next one.

AMCilk runs multiple Cilk jobs in a single runtime system. Recall that (Section 2) the original Cilk runtime system is designed to run a single job,

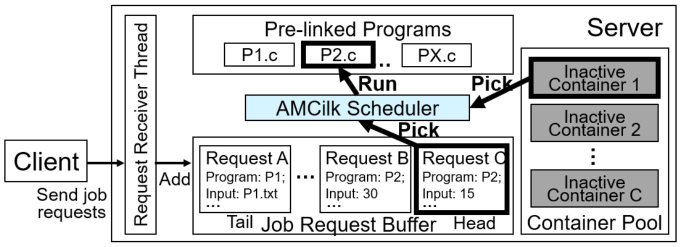


Figure 1: AMCilk's Client-Server Architecture.

where the `main` function of the job's executable initializes the runtime and calls `cilk_main` as an entry into the user code. In contrast, the AMCilk runtime system is pre-initialized as a server and sets up the basic data structures needed to execute jobs. The parallel programs are pre-compiled and pre-linked with the runtime system and have their `cilk_main` functions. To run multiple jobs, the server runs each job within a data structure called a ***container*** which contains all the metadata required to run Cilk jobs. Since jobs arrive and leave online, the number of active containers changes over time. However, creating a container from scratch is relatively expensive, so AMCilk creates a pool of containers at initialization and reuses the containers. When a new job arrives, the server selects an inactive container and calls the appropriate `cilk_main` function to start executing the job. When all containers are busy[1], any new arriving job is buffered. When a container becomes available, it picks a job from the buffer in a FIFO order.

3.2. Policy-Customization Interface

AMCilk provides an interface that allows the system administrator to customize the policy for allocating cores, cache, and memory bandwidth between concurrent jobs. We provide some useful allocation policies "out of the box" — these are the policies we used in our case studies described in Section 6, namely (1) DREP; (2) ELASTIC_RT; and (3) PARALLELISM_FB. System administra-

[1]This case rarely happens, since we use a large pool — we set the number of container to be equal to the number of cores used for executing jobs.

tors can design their own policies and implement them using a simple interface provided within AMCilk.

The reallocation decision interface is event-driven. AMCilk provides four events: (1) `START_JOB`; (2) `EXIT_JOB`; (3) `TIMER`; (4) `REQUESTED`. When any event happens, the `job_scheduler(e)` function is called — this is the function that the system administrator implements in order to design their own core-allocation policy. The `job_scheduler(e)` has an argument `e` indicating which event triggerred the current function call (to the `job_scheduler(e)`). The system administrator can use this argument to distinguish different events and define appropriate response to different events (or ignore some events).

Within this function implementation, the system administrator can use predefined functions to both get information about the current state of the runtime system and to change the allocation of cores, memory bandwidth and cache. In general, to perform core-reallocation, one must (1) analyze the runtime information; (2) make a core-reallocation decision; (3) assign cores to jobs. AMCilk collects the runtime information in the backend, and the interface exposes the information to the system administrator, like the number of running jobs, the number of available cores and the current scheduling state showing which core belongs to which job. The interface also exposes in-depth runtime details, like the number of cycles when each core was working vs. stealing in the previous interval. Within `job_scheduler(e)`, the system administrator can call various functions to access this information and use this information to make scheduling decisions. The scheduling decisions can be communicated to the AMCilk scheduler by using setter functions — for example, AMCilk defines `core_id` to denote a core and `container_id` to denote a container, and the system administrator can use `give_core_to_container(core_id, container_id)` to allocate a core to a container. AMCilk will then automatically enforce this reallocation using a safe, responsive, and low overhead preemption and core reallocation method described in Section 3.4.

Using the policy-customization interface can greatly simplify the system implementation. For example, Figure 2 shows an implementation of DREP sys-

```
//DREP implementation with AMCilk interface
void job_scheduler(enum PLATFORM_SCHEDULER_TYPE e) {
    container_id_t new_p, stop_p, tmp_p;
    if (e==NEW_PROGRAM) { //when a new job begins
        new_p = get_new_program_container();
        for (int i=2; i<get_total_num_core(); i++) { //randomly get cores for the new job
            int rand = util_random_selector(get_num_running_job());
            //rand==0 with probablity of 1/get_num_running_job()
            //rand===0 when get_num_running_job()==1 (the new job is the first job).
            if (rand==0) give_core_to_container(new_p, i); //give core i to container new_p
            //by default, a new container occupies no core unless we do give_core_to_container
        }
    } else if (e==EXIT_PROGRAM) { //when a job completes
        stop_p = get_stop_program_container();
        for (int i=2; i<get_total_num_core(); i++) {
            if (container_use_core(stop_p, i)) { //core i was used in the stop program
                tmp_p = random_pick_unfinished_container(); //give the core i to a random job
                if (tmp_p!=NULL) give_core_to_container(tmp_p, i);
            }
        }
    }
}
```

Figure 2: Implementation of DREP scheduler via AMCilk policy-customization interface.

tem [10] in AMCilk with the policy-customization interface. DREP is a system designed for online scheduling of multiprogrammed parallel jobs to minimize average flow time. Basically, when a new job arrives, for each core, the DREP scheduler gives this core to the new job with probability of $p = 1/n_t$ where n_t is the number of unfinished jobs. When a job finishes, for each core, the scheduler randomly picks an unfinished job to give that core. As shown in Figure 2, the system is implemented in simply 18 lines (without comments), and no system details are needed in the implementation.

AMCilk provides a similar interface to customize cache partitioning and memory bandwidth allocation policies. Again, the system administrator can access runtime information via the interface, like cache misses, and the administrator can use the interface to allocate cache blocks and set maximum memory bandwidth usage of each container. Note that AMCilk is extensible, and system experts could develop their own runtime information collectors and events under our scheduling framework.

3.3. Decentralized AMCilk Scheduler

The AMCilk scheduling framework enables concurrent running of multiple parallel jobs and reallocates computing resources, including core, last-level cache, and memory bandwidth, between jobs according to the customized scheduling policy. Figure 3 zooms into the architecture of the scheduler itself. The Runtime Monitoring Module keeps track of the runtime information, such as core utilization, of running jobs (step1) and sends the information to the Resource Allocation Module (step2). The Resource Allocation Module decides how many resources should be allocated to each job based on the scheduling policy (which was implemented by the system administrator) and sends the decision to the Resource Enforcement Module (step3), which fulfills the allocation to jobs via their containers (step4).

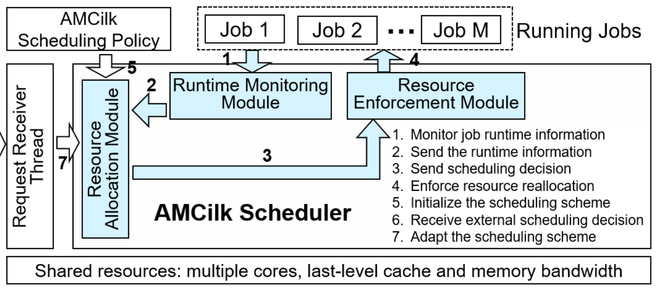

AMCilk Scheduling Policy
Job 1
Job 2
Job M
Running Jobs
Request Receiver Thread
Resource Allocation Module
Runtime Monitoring Module
Resource Enforcement Module
AMCilk Scheduler
1. Monitor job runtime information
2. Send the runtime information
3. Send scheduling decision
4. Enforce resource reallocation
5. Initialize the scheduling scheme
6. Receive external scheduling decision
7. Adapt the scheduling scheme
Shared resources: multiple cores, last-level cache and memory bandwidth

Figure 3: AMCilk scheduling framework.

The AMCilk scheduling framework provides interfaces that allow the system administrator to easily customize the scheduling policy for its application scenario in the Resource Allocation Module (step 5). Furthermore, AMCilk exposes an interface that allows external systems to control the resources used by AMCilk via sending the demand to the request receiver thread (step 6), which invokes the AMCilk scheduler to enforce the allocation demand (step 7).

To perform the cache partitioning and memory bandwidth allocation decided by the scheduling policy, Resource Enforcement Module calls the interfaces provided by third-party infrastructures. For example, Intel RDT [24] that we use in this work provides interfaces for allocating last-level cache and memory bandwidth to core groups. So the AMCilk scheduler groups the cores assigned to each running job and calls Intel RDT to perform the allocation to the core groups.

To support concurrent execution and dynamic core allocation of multiple parallel jobs, AMCilk decouples the concept of the core (physical processing unit) and the worker (software abstraction of a core). For a machine with p cores (excluding the core dedicated to the request receiver thread), AMCilk creates p workers (threads) *for each container* dedicated to a job, and each of these workers is pinned to a different core. Hence, each core has multiple workers, one for each container. The Resource Enforcement Module ensures that each running job occupies a disjoint set of cores according to the core allocation decision, by activating at most one worker on each core. An example snapshot is shown in Figure 4.

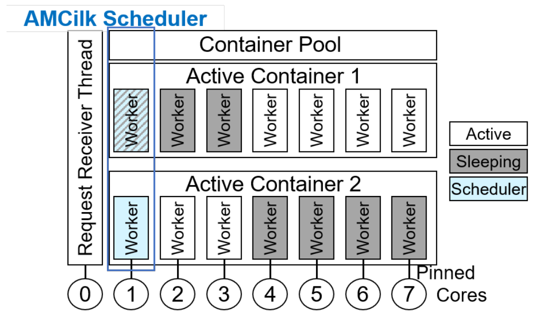


Figure 4: Runtime snapshot. Container 1 is allocated with 4 cores with 4 active workers, while Container 2 is allocated with 2 cores.

For each job, the cores allocated to this job must complete its work using a modified work-stealing scheduler that we augmented to support three novel functionalities needed by the AMCilk scheduling framework: decentralized scheduling, core reallocation, and work resumption. We explain the decentralized scheduling here and the other two mechanisms in the next subsections.

Although a core is dedicated for the AMCilk scheduler (leaving $p-2$ cores allocated by the scheduling policy for executing jobs[2]), instead of a dedicated centralized thread for the scheduler, each container handles its own allocation by setting its worker 1 be a dedicated scheduler worker when ***starting a job***

[2]One core is dedicated for the AMCilk scheduler, and one core is dedicated for receiving job requests, leaving $p-2$ cores to execute jobs.

and ***removing a job***. Specifically, to start a new job from the request buffer, a container from the container pool is activated by waking up its worker 1. This worker prepares all the necessary data structures for this job and decides which cores should be allocated to this job, based on the customized scheduling policy provided by the system administrator. This will trigger reallocation so that these cores are allocated to this new job. At this point, the `cilk_main` function of this new job is called and the job execution begins. When a container completes a job, one random worker returns from the `cilk_main` and enters the runtime. This worker will activate the worker 1 of its container before putting itself to sleep. Then this worker 1 will clean up the data structures for this job, trigger the core reallocation per the scheduling policy, and inactivate itself (and this container) once done. If other scheduling events occur, for instance, due to external triggers or timing triggers, a dedicated thread pinned on core 1 for the AMCilk scheduler will wake up to make the new scheduling decision and trigger the core reallocation.

3.4. Responsive and Low-Cost Core Reallocation

During job execution, the scheduling policy may decide to change the core allocation of jobs, i.e., some job(s) must give some of their cores to other jobs, and some job(s) may reclaim the cores it gives out in the previous scheduling, triggering AMCilk's core reallocation mechanism. Reallocating a core x that is currently used by job a to job b involves two procedures: putting the running worker of job a on core x to sleep and waking up the worker of job b on core x. The second procedure can be achieved by simply sending a signal to wake up the corresponding thread. If the woken-up worker has some work on its deque, then it resumes working on its deque. Otherwise, it immediately starts stealing.

The first procedure, namely ***worker preemption*** where a worker stops working and goes to sleep, is the key to core allocation. This operation must be *safe* (i.e., we don't want to preempt a worker while it is holding a lock, for example), *responsive* (i.e., given a reallocation decision, the worker should go to sleep as soon as possible), and *low overhead* (i.e., its overhead should have minimal impact on performance).

There are a few options for implementing worker preemption. One possibility is to use the priority mechanism of the operating system (OS). Say the scheduling policy decides to allocate a core x to job b while it is currently allocated to job a. The containers for both jobs have a thread pinned to core x, so the scheduler could increase the priority of the job b's thread on core x and decrease the priority of the job a's thread. One disadvantage of this method is that this context switch has high overheads. More importantly, it is difficult to ensure correctness and performance since the thread of job a might be holding a lock when it is put to sleep by OS, causing it to block other threads from doing work.

To ensure that the thread is put to sleep when it is safe to do so, another approach, taken by Agrawal et. al [10], is to allow worker preemption only when the worker attempts to steal. In particular, on receiving the decision that a worker w must be put to sleep, the corresponding work-stealing scheduler waits until worker w has no work on its deque and is about to steal. At this point, it puts the thread to sleep. This is, in some sense, the safest and easiest place to implement a preemption within the runtime system since, as described in Section 2, the worker is not working on anything and does not have any work on its deque. However, this mechanism would not be very responsive since the worker may not steal for a long time. Therefore, the time between the occurrence of the decision that some core x should be moved from job a to job b and the time when job a actually puts its worker on core x to sleep can be huge.

In contrast, we employ the middle road and use the exception mechanism of the Cilk runtime system (described in Section 2) to implement preemption. When the AMCilk scheduler decides to take away core x from a job, it sets the exception pointer (E) of the worker w on core x to a large number. When worker w finishes its current frame, it finds that $E > T$ and jumps to the exception handling routine. This routine then sets up the state indicating that worker w is now inactive and puts the associated thread to sleep. It is important to note that the preempted worker may still have work on its deque but it may never be woken up again, so efficient work resumption, explained in Section 3.5, is needed to complete the work left on this deque by other workers of the same job.

Our design choice for worker preemption is reasonably responsive since it implements preemption at frame (function) boundaries — the worker to be preempted is preempted as soon as it finishes the function it is currently executing. For most fine-grained parallel code, the individual functions are reasonably small. In addition, since the preemption is handled by the runtime system, it can ensure that the thread is not holding locks when it is preempted.

3.5. Efficient Work Resumption Mechanism

As discussed above, since AMCilk implements preemption at frame boundaries, a worker w of job a can go to sleep while there is still work (frames) on its deque. This work must be resumed by some workers of job a so that job a can successfully complete. To facilitate work resumption, each worker has a status field. Before an `active` worker w goes to sleep, it first checks if its deque has any remaining work. If there is remaining work, it marks its status as `inactive_with_work`; otherwise, it marks its status as `inactive_without_work`.

All workers of the job are stored in an array of size p, where p is the number of (active and inactive) workers. This array is sorted to store all the `inactive_with_work` workers at the beginning and the `active` workers in the middle, followed by all the `inactive_without_work` workers. We also maintain two auxiliary pointers pointing to the last location storing an `inactive_with_work` worker and the first location storing an `inactive_without_work` worker, as shown in Figure 5.

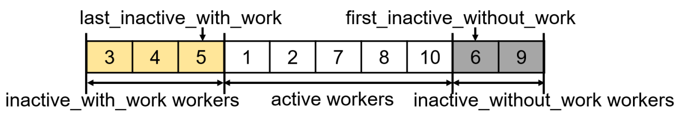

last_inactive_with_work
first_inactive_without_work
3
4
5
1
2
7
8
10
6
9
inactive_with_work workers
active workers
inactive_without_work workers

Figure 5: A job's worker array, storing all its workers sorted in a way that makes it easy for active workers to mug and steal.

In addition to the above data structures, we implement the key operation, called ***mugging***, for efficient work resumption. Recall that, in original work-stealing, when a worker runs out of work, it randomly picks a victim and steals work from the top of the victim's deque. In AMCilk, when an active worker of job a runs out of work (i.e., its deque is empty), it first checks the worker array to see if there are any `inactive_with_work` workers. If so, it picks one as the victim and mugs the victim's worker by swapping the victim's nonempty deque with its own empty deque. It then moves the victim to the last portion of the worker array (the `inactive_without_work` portion, since this worker now has an empty deque) and updates both auxiliary pointers. Once there is no `inactive_with_work` worker, regular work-stealing among the active workers is resumed efficiently by storing the `active` workers contiguously. With the help of the two auxiliary pointers, AMCilk avoids the unsuccessful steal attempts from sleeping workers with empty deques.

Our design for the work resumption mechanism has the advantage that it maintains the theoretical and practical performance guarantees provided by work-stealing [1]. Intuitively, these guarantees depend on the fact that if there are d total deques for a job, then d random steal attempts will reduce the critical-path length of the job with high probability. However, if we have more deques, we need more steal attempts to make progress. In AMCilk, if there are sleeping workers with nonempty deques, we prioritize making their deques empty and never steal from sleeping workers with empty deques. Therefore, if the job has x active workers, this design only needs x steal attempts to reduce the critical-path length — in systems with many jobs, the number of cores may be much larger than x and this design is efficient. The theoretical guarantees provided by some multiprogrammed application scenarios [13] depend on this mechanism.

4. Support for Shared Cloud Environments

Thus far, we have presented AMCilk as though it fully occupies the entire physical machine. However, in some shared environments such as clouds, the AMCilk process may share the resources, such as cores and memory bandwidth, with other processes. In these scenarios, an external scheduler, like a cluster scheduler, should be able to dynamically control the resource occupied by the AMCilk process.

This section introduces the key features of AMCilk that allow the AMCilk system to run in such shared cloud environments. In particular, AMCilk runtime system supports: (1) Subscription of runtime information — an external scheduler can monitor the runtime information of each job running in the AMCilk process; (2) Resource occupancy control — an external scheduler can set the upper bound of resources used by the entire AMCilk process (AMCilk will still control how to allocate resources to each of its own jobs based on the mechanisms described in the previous section); (3) Admission control — the external scheduler can set the maximum buffer length for arriving jobs for AMCilk, thereby providing admission control for AMCilk jobs. As a result, the external scheduler can gather runtime information for the AMCilk process, use this information to decide how much resources to provide the AMCilk process, and also put constraints on the buffer length and the resource occupancy of the entire AMCilk process. Note that this external scheduler is outside of AMCilk and AMCilk has no control over it.

4.1. Subscription of runtime information

There are three components in the runtime information provided by AMCilk: (1) hardware usage, including processors, memory, last-level cache, off-chip memory bandwidth. This is the basic information showing how busy the AMCilk system is. However, this hardware usage information is often misleading. Recall that each parallel job running in the AMCilk is executed by a work-stealing scheduler. If a job is allocated more processors than its parallism, some worker of the job can be busy for stealing attempts. In this case, although the processor usage is high, this job does not need as many as the processors it is allocated. Thus, it may be useful for the external scheduler to know (2) the internal runtime information for each job, such as working cycles and stealing cycles to know the actual resource utilization for each job. Furthermore, the external scheduler need to distinguish whether the AMCilk system is overloaded when the resource utilization is high. Therefore, AMCilk exposes (3) the length of request buffer to the cluster scheduler.

AMCilk also provides interfaces which allow system designers to define any metric based on the runtime information, and AMCilk system automatically exposes the metric to the external scheduler via the subscription. For example, if the external scheduler needs to know the current average flow time of jobs, the system designer sets the AMCilk to gather the flow time of each jobs and sets a time window to calculate the average within the window. Currently, AMCilk supports the subscription of average flow time, maximum flow time, minimum flow time and percentile latency. System designers can either choose one of these metrics or design their own metric — AMCilk will then calculate this metric and provide the information through to the subscriber.

The subscription is implemented following publish–subscribe pattern, which decouples the AMCilk and the external scheduler and reduces data size in communications. Since the external scheduler may be located at a different machine, the runtime information should be visible by a different machine. The AMCilk system runs as a publisher which periodically updates the runtime information to keep updated. Noting that the external scheduler may not need all runtime information that AMCilk exposes. Thus, the runtime information is organized in key-value pairs, where keys are the name of resources, like ”stealing_cycle”. The runtime information is updated in a transactional manner — either all keys are updated or no key is updated. The external scheduler runs as a subscriber. By providing the keys of interest, the external scheduler can get the specific values for those keys from the latest runtime information on demand.

4.2. Resource Occupancy Control and Admission Control

AMCilk runtime exposes interfaces to the external scheduler to control the occupancy of three resources: (1) the number of processors; (2) last-level cache; (3) off-chip memory bandwidth. When the external scheduler decides to adapt the resources occupied by the AMCilk process, the AMCilk process leverages the mechanisms we described in Section 3 to adapt to these new resources. In particular, once the AMCilk runtime receives a request to adapt resource occupation from the external scheduler, the internal AMCilk scheduler runs and changes the allocation of its jobs based on its current policy. Once the enforcement is done, AMCilk notifies the external scheduler. Since the resource reallocation of jobs in the AMCilk is fast, the external scheduler can adapt the resource occupancy of AMCilk with low latency. Note that given a change in resources, the actual allocation of resources to the jobs running under AMCilk is based on the scheduling policy provided to AMCilk via the AMCilk interface. Therefore, when running in a shared environment, system designers must provide scheduling policies which can intelligently respond to resource changes to get maximum benefit from this feature.

Recall that the AMCilk runtime system runs as a server which receives job requests from clients. Once a job request is received, this request is appended to a request buffer. If the resources owned by the AMCilk runtime is insufficient to handle the request frequency, the number of buffered job may increase unboundedly causing long delays in processing of jobs. In order to avoid this, we allow the external scheduler to do ***dynamic admission control*** where it can change the AMCilk's maximum buffer length by communicating with the AMCilk server. Then the AMCilk still admits the jobs in a first-come-first-serve order, but it rejects any requests which cause the buffer size to grow larger than the maximum buffer length.

5. Evaluation

We evaluate AMCilk performance using two types of benchmarks. In this section, we try to understand the efficiency of *AMCilk implementation* by quantifying the system overhead and examining the advantage of cache and memory bandwidth allocation functionalities. In the next section, we will try to understand the ***impact*** of AMCilk on multiprogrammed applications to see if AMCilk can provide a performance boost for their application-specific metrics.

We conducted the evaluation on a machine with two 2.40GHz Intel Xeon Gold 6148 Processors and 754GB memory. Each processor has 20 physical cores with 27.5 MB L3 Cache, and the system has 40 physical cores in total. The two processors support Intel RDT which provides capabilities for cache and memory allocation and monitoring. The Linux version is 4.15.0. AMCilk uses Intel(R) RDT Software Package[3] to control the hardware-level cache partitioning and memory bandwidth allocation. In the experiments, we disabled hyperthreading. Two cores are reserved for the request receiver and the AMCilk scheduler, respectively; the remaining 38 cores are used to execute jobs.

5.1. System Overhead

We first conduct experiments to quantify the time costs of the four core functionalities that AMCilk promises (as discussed in Section 3): (1) starting a job; (2) removing a job; (3) core reallocation; (4) work resumption.

Experimental Design. We measured the overhead by instrumenting each individual operation and running a latency-sensitive application [15]. We develop a simple client to generate the workload. The workload includes a series of online requests. The work of each request is random and follows ***Bing Search Server Request Work Distribution*** [15]. Each request is computationally intensive and each request is parallelized by using parallel-for loops. Note that since we evaluate the time span of scheduling actions on parallel jobs, the result will not

[3]The package is open source and maintained at https://github.com/intel/intel-cmt-cat.

change much between different computations since the individual functions are reasonably small for most fine-grained parallel code. We use Poisson distribution to decide inter-arrival time between requests. By configuring the average number of requests that arrive per second, we can set the utilization of the system. We wanted to examine whether the load affects the overhead and varied the total load of the application, i.e., machine utilization from 60% to 90%, by changing the average number of requests arrived per second. We observed that the machine utilization has a very small impact on the overhead, so we only report the results for a machine utilization of 75%. The experiments were run long enough, and we measured the time to run each operation for 100,000 times and report the mean and standard deviation. To improve the readability of boxplots, we randomly sample 1,000 of the 100,000 measurements to draw Figure 6. We found 15 outliers in the measurements in total, and we removed the outliers when calculating the maximum latency (and 50th/95th/99th/99.5th percentile).

Evaluation Results. As explained in Section 3.3, ***starting a job*** includes taking a job from the request buffer, setting up the container for this job, and allocating resources to this job. In our evaluation, this functionality takes 295μs on average with a standard deviation of 489μs. Half of the measurements take no more than 291μs, and almost all measurements are no more than 440μs. In particular, the 95th percentile is 369μs, and the 99th percentile is 414μs, and the 99.5th percentile is 440μs, and the maximum measurement is 8834μs. Note that allocating resources to a new job often involves reallocating cores, so this time cost is dominated by core reallocation (272μs). Recall that in our design, containers are created at AMCilk system initialization and are reused upon job arrivals. We evaluate this design choice with an experiment where we create containers from scratch every time a new job arrives. As expected, always creating containers is significantly more expensive with a mean overhead of 4379μs, due to the cost of creating pthreads for workers and allocating and (more importantly) initializing the data structures for the closures, frames, and

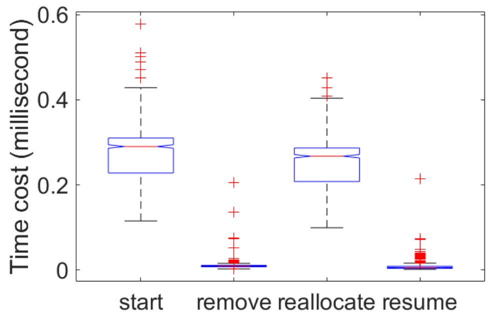


Figure 6: Time cost of key funcationalities

fibers that the runtime system uses.

Removing a job involves deallocating cores (and other resources) of the completed job and releasing the container back to the container pool. This functionality costs 10.0μs on average with a standard deviation of 21.5μs. Half of the measurements take no more than 8.71μs, and almost all measurements are no more than 197μs. In particular, the 95th percentile is 12.8μs, and the 99th percentile is 61.2μs, and the 99.5th percentile is 197μs, and the maximum measurement is 457μs. Removing a job takes a significantly shorter time than starting a job because it only deallocates cores. The reallocation of these cores is either performed in starting a new job or performing the core reallocation for the active jobs based on the scheduling policy.

Core reallocation includes deciding the resource allocation for jobs according to the customized scheduling policy and enforcing the decision. Of the 272μs average overhead (std. 480μs). Half of the measurements take no more than 267μs, and almost all measurements are no more than 401μs. In particular, the 95th percentile is 345μs, and the 99th percentile is 384μs, and the 99.5th percentile is 401μs, and the maximum measurement is 8812μs. On average only 17.5μs is spent on making the decision, so enforcing the decision introduces the major overhead. Recall that enforcing the decision involves putting a worker to sleep for one job and activating a worker for another job. Activating a worker costs 57.5μs, while putting a worker to sleep costs 85.2μs. The latter operation takes more time because it includes waiting until the worker reaches the frame boundary. Obviously, this overhead would be significantly higher if the worker has to reach a steal boundary instead.

Work resumption starts when a worker with a non-empty deque goes to sleep and ends when another worker successfully jumps to the user code after finding and mugging this nonempty deque of a sleeping worker. This functionality costs 7.20μs on average with a standard deviation of 7.50μs. Half of the measurements take no more than 5.54μs, and almost all measurements are no more than 53.2μs. In particular, the 95th percentile is 16.4μs, and the 99th percentile is 32.8μs, and the 99.5th percentile is 53.2μs, and the maximum measurement is 220μs. For resuming the work of inactive workers, we could let a thief steal from the victim's deque one frame at a time, instead of mugging the entire deque. To verify our choice of mugging, we measure the overhead of both operations. We observe that a mugging operation costs 0.363μs (std. 0.204μs), which is actually less than the cost of 1.44μs (std. 3.00μs) of a successful steal. This result is as expected since a successful steal involves taking multiple locks, manipulating data structures, and promoting the child frame to make it ready for a potential future steal. Mugging is much simpler; we just grab a lock and change some pointers around. Therefore, mugging not only reduces the number of active deques, but also has a smaller overhead.

The experiments show that all operations have small average costs, but their variations are not negligible. The variations come from contention, instead of noise. In particular, the measured time includes the operation of locking data structures before modifying them. Therefore, the cost is higher when we have to wait on the lock. Additionally, some optimizations — there are fast paths and slow paths depending on the particular situation — also lead to variation.

5.2. Cache partitioning and memory bandwidth allocation

Since the overheads of cache and memory bandwidth allocation of AMCilk are the same as Intel RDT, we do not measure these costs. Instead, we demonstrate their capability of reducing interference in a scenario where data-intensive parallel jobs co-run with streaming applications.

Experimental Design. We use a `parallel_sort` program, which takes an array as the input and returns the sorted array, as the data-intensive job. We randomly generate an array with 50,000,000 64-bit elements. We use AMCilk to run 4 such jobs concurrently, where each job is allocated with 4 cores. We also design a parallel streaming job that repeatedly loads data from memory, modify the data, and store the data into the memory. When co-running with the 4 data-intensive jobs, this streaming job is allocated with the remaining cores in the platform. We measure the running time of the 4 data-intensive jobs in 4 cases: (1) only running the 4 jobs; (2) co-running the 4 jobs with the streaming job; (3) partitioning the cache between the 4 jobs and the streaming job; (4) restricting the memory bandwidth usage of the streaming job. For each case, we record the running time for 1,000 times.

Evaluation Results. As shown in Table 1, when co-running with the streaming job, the data-intensive job‘s running time increases by 13.4%. With cache partitioning (CP), the job running time reduces by 2.8%. With memory bandwidth allocation (MBA), where we restrict 10% for the streaming job, the job running time decreases back to the time of running alone. This simple experiment shows that cache and memory bandwidth allocation can effectively reduce interference between jobs and providing this functionality is crucial to enable the design of efficient multiprogrammed systems using AMCilk.

Table 1: Running time of data-intensive jobs

	(1) Alone	(2) Co-run	(3) Co-run+CP	(4) Co-run+MBA
Mean (second)	1.86	2.11	2.05	1.87
Std. (second)	0.0264	0.0600	0.0360	0.0286

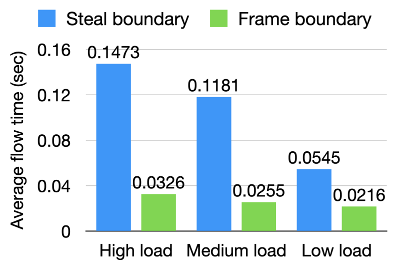
6. Case studies

Multiprogrammed applications are ubiquitous. In this section, we present five concrete examples of multiprogrammed application scenarios with differing needs. We implemented all five scenarios using AMCilk on the hardware used in Section 5 and ask the following question: does the AMCilk implementation provide improved performance to these applications for the criteria that these applications care about — in other words, do the responsive and low-overhead core reallocation and cache partitioning and memory bandwidth allocation provide a measurable impact on the application-specific performance of these applications?

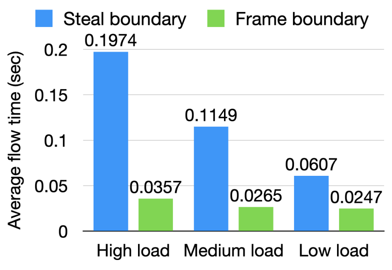
6.1. Online Scheduling to Minimize Average Flow Time

In the context of ***interactive services***, users send requests to the service, and the service must process the requests while optimizing some service-wide performance criteria. We consider the ***online*** scenario where the jobs (computation done to satisfy requests) are parallel and the service does not know the characteristics of the jobs (such as their running times or arrival times). One of the most commonly used quality-of-service metrics is the ***average flow time*** of all jobs, where the flow time of a job is the elapsed time between the job's arrival time and its completion time.

Several scheduling algorithms have been designed and theoretically analyzed for minimizing average flow for parallel jobs [7, 8, 9, 10]. The only one that has been implemented is the ***Distributed Random Equi-Partition*** (***DREP***) algorithm [10], which was shown to have good performance theoretically and practically. Given the DREP scheduler, when a job arrives at time t, each processor decides to give itself to the new job with probability $1/n_t$ where n_t is the number of unfinished jobs at time t. When a job completes, each processor assigned to that job randomly picks an unfinished job and gives itself to the picked job. Agrawal et al.'s implementation [10] shows their scheduler design has strong practical performance, but the performance can significantly increase



(a) Bing Search Workload



(b) Finance Workload

Figure 7: Average flow time with Bing Search Workload (left) and Finance Workload (right). Flow time is measured as the time span from the moment when a request arrives at the system to the moment when the system completes the request. AMCilk implements preemptions at frame boundaries and this experiment compares the result to preemptions compared at steals as proposed by prior work.

further if the DREP scheduler is implemented based on AMCilk system, due to a faster core reallocation mechanism. In particular, in Agrawal et al.'s implementation [10], preemption only occurs at steal boundaries (as described in Section 3). When a new job arrives, the DREP scheduler allocates certain cores to it which were allocated to other jobs. The cores only stop working on their current jobs and start working on the new jobs when their deque becomes empty and they try to steal. In contrast, AMCilk implements preemptions at frame boundaries, leading to more responsive reallocations.

We compared the frame-boundary preemption of AMCilk and the steal-boundary implementation[4] using the workload distribution from real applications: *bing search workload* and *finance server workload* [15]. For each workload, we vary the average number of jobs arrived per second to generate three different system loads: *low*, *medium*, and *high loads*, where the average system utilizations are approximately 60%, 75%, and 90%. For each setting, we randomly

[4]The implementation in [10] was based on the Cilk Plus runtime system. For a fair comparison with AMCilk, we implemented their steal-boundary preemption in the Cilk-based Cheetah runtime system.

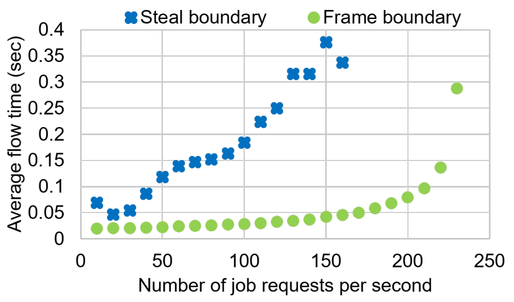


Figure 8: Average flow time with different request frequency with Bing Search Workload. Flow time is measured as the time span from the moment when a request arrives at the system to the moment when the system completes the request.

generate 100,000 jobs and record the average flow time. Figure 7 shows the results for the bing search workload and the finance workload. The result indicates that the frame-boundary implementation reduces the average flow times by 60-70% compared to the steal-boundary implementation for the bing search workload, and the reduction of average flow time is even larger for the finance workload. Figure 8 compares both systems by increasing the job arrival rate of the Bing workload. We can see that AMCilk supports the job arrival rate of up to 230 jobs per second without being overloaded (where overloading is indicated by having the average flow time increase unboundedly as time passes) while the frame-boundary implementation supports at most 160 jobs per second — an improvement of 43.8%, indicating that fast preemption can indeed lead to measurable impact on service-level performance for this application.

6.2. Elastic Parallel Real-Time Scheduling

In cyber-physical systems, such as autonomous vehicles and robotics, sensors periodically collect environment data, and the computing component must process the data to calculate the control demands by the end of the period. Abstractly, such a system contains a set of ***real-time tasks*** — each task τ_i is defined by a tuple $\{C_i, T_i\}$, where C_i is the maximum execution requirement of each ***job*** of the task and the task can release jobs with a period (minimum inter-arrival time) of T_i. In the simplest scenario, each job has a deadline of T_i

— it must complete in T_i time after it is released.[5]

We are interested in ***parallel real-time tasks*** where the jobs of real-time tasks may contain internal parallelism — in particular, we focus on ***elastic real-time tasks*** [22]. In this model, tasks can change or tolerate a change in their utilizations $U_i = C_i/T_i$ (by changing either C_i or T_i or both) due to the change in the physical system — for instance, if the system enters a less stable state and requires a more expensive or faster control algorithm. The tasks that can increase its utilization are ***demanding tasks***. To satisfy the utilization increase of a demanding task, additional cores must be given to this task (to meet its deadline) by reducing the cores given to the non-demanding tasks. Orr et al. [22] established an ***elastic scheduling algorithm*** to calculate the core allocation for all tasks when a demanding task changes its demand — the details are complex and not relevant to this discussion — the key is that the platform running these applications must be able to reallocate cores among jobs due to external stimuli.

Orr et al. [22] conducted experiments on elastic scheduling using OpenMP; however, they did not have access to a platform with responsive and low-cost core reallocation mechanism while jobs were running. In their system, after the elastic scheduler computes a new allocation, a demanding task gets additional cores only after the currently running jobs of non-demanding tasks have completed. Hence, the delay between demanding more cores and actually getting these cores depends on the other tasks' period. In contrast, AMCilk allows reallocation at any time during the job's execution, so the demanding tasks get additional cores much more quickly.

We demonstrate the benefit of fast reallocation on the performance of elastic task systems by running a simple experiment with 2 tasks. Both tasks calculate the 42nd Fibonacci number. Note that the performance in this application scenario depends on the latency of core reallocation, which is evaluated in Section 5. We vary Task 1's period from 10 to 600 milliseconds, while fixing Task

[5]In the general setting, the deadline may be different from the period.

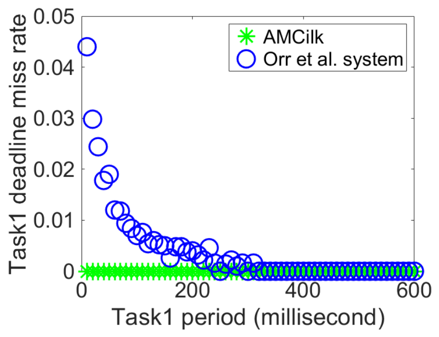


Figure 9: Elastic scheduling with task 1 and task 2. The period of task 1 reduces to 1/3 of its original value at an arbitrary time while the period of task 2 remains the same. We show the deadline miss rate (vertical axis) of the task 1 with different original task 1's period (horizontal axis).

2's period as 50 milliseconds. For each setting, we run task 1 for 1000 iterations. We randomly select 10 iterations to let task 1's period be reduced to 1/3 of its original value and let this change lasts for a random length from 1 to 10 iterations. Figure 9 shows task 1's deadline miss rate — the number of jobs missing their deadlines divided by the total number of jobs. In real-time systems, the goal is to not miss any deadlines. Since AMCilk allows for fast core reallocation regardless of tasks' period, task 1 never misses any deadlines. In contrast, the deadline miss rate of Orr et al.'s system depends heavily on the periods of the two tasks. As task 1's period gets smaller (compared to task 2's period), task 1 misses more deadlines.

The ability of AMCilk to reallocate cores with predictable delays that are independent of job periods is a huge advantage for real-time systems. The goal of real-time system is to provide an *a priori* guarantee on the timing properties of the system. AMCilk makes it easier to provide such guarantees, since the predictable delays can be incorporated into the a priori timing analysis, while this is harder to do so when the delay depends on the job characteristic.

6.3. Adaptive Scheduling Using Parallelism Feedback

Fine-grained multithreaded jobs, such as those written using Cilk, can change parallelism as they execute. Thus, statically allocating a fixed number of cores when a job arrives is often inefficient, as the number of cores that can be used by the job depends on whether it is in its low- or high-parallelism phases. Thus, Agrawal et al. [13] proposed an adaptive scheduling strategy that dynamically adapts the number of cores allocated to a job based on an estimate of the job's dynamic parallelism. Given a job, this scheduler periodically collects the number of steal-cycles and work-cycles and mug-cycles on each processor allotted to the job in runtime and uses this information to decide whether the job needs more processors or whether a program occupies too many processors. The scheduler dynamically adapts the number of processors of the program accordingly. While the details are not relevant, this scheduler monitors all jobs' runtime characteristics and periodically changes the core allocation based on these characteristics.

We implemented this adaptive scheduling algorithm using AMCilk. This implementation demonstrates an interesting feature of AMCilk that the previous examples don't. For DREP, the core allocation changes only when new jobs arrive or when jobs complete. In elastic scheduling, core allocation changes due to external signals. In adaptive scheduling, AMCilk monitors the internal characteristics of the jobs and changes the allocations based on these characteristics.

We evaluate the AMCilk implementation of adaptive scheduling using a simple experiment with 2 jobs that change their parallelism frequently: each job repeatedly switches between high- and low-parallelism phases for 10 times, where the phase of one job is opposite to the other job. In the high-parallelism phase, the job has one large parallel for-loop with 12,800,000 iterations, while in the low-parallelism phase, the job has 4000 small parallel for-loops, each with 100 iterations. AMCilk should capture the switch between the high- and low-parallelism phases of the two jobs quickly and adapt the core allocation of the two jobs responsively. As a result, the running time for both jobs should be smaller than the static partition case for better core utilization.

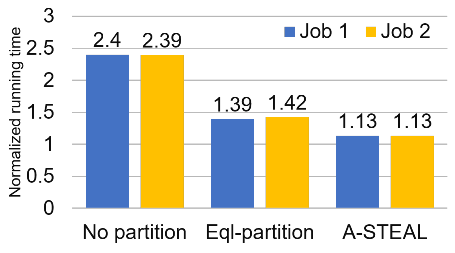


Figure 10: Adaptive scheduling. We run two jobs at the same time with three schedulers (horizontal axis): (1) No partition where each job occupies all cores, and so every core has two jobs running; (2) Eql-partition where each job exclusively uses half of number of cores in system; (3) A-STEAL where each job gets cores on demand according to its runtime parallelism (our scheduler).

According to [13], the period of core adaptations should be long enough to amortize the time for core reallocations. Since AMCilk core reallocation is 272μs, we gradually decrease the period from 10ms to 0.5ms and explore the value of the period such that the running time of the two jobs is minimal. Finally, we set the period as 1ms in our experiment.

There is no existing implementation of adaptive scheduling, so we compare against static allocations. We measure the running times of the jobs and normalized them using the running time of 1.65 seconds when each job run individually on all (38) cores. As shown in Figure 10, if we do not partition the cores and let the two jobs share the 38 cores, their running times become 2.4 times of their solo running times. If we statically and equally partition the cores, i.e., giving each job 19 cores, they complete in 2.32 and 2.34 seconds. Using the AMCilk implementation of adaptive scheduling (with a reasonable setting of parameters), the two jobs complete in 1.86 and 1.87 seconds — 19.8% and 20.1% reductions over equal-partition. This is because our implementation is able to monitor the parallelism of jobs and give fewer cores (about 8 cores) to the job in the low-parallelism phase and more cores (about 30 cores) to the job in the high-parallelism phase. More specifically, when a job changes from low-parallelism to high-parallelism, it experiences 8 times of getting more cores decided by the adaptive scheduling policy, which takes 47.9 milliseconds in total. The functionalities provided by AMCilk makes it possible to implement the adaptive scheduling efficiently for multiple parallel jobs with dynamic parallelism.

6.4. Co-scheduling Throughput and Tail-sensitive Jobs

The previous experiments have explored the impact of the fast core-reallocation ability of AMCilk. The final experiment explores the impact of its cache and memory bandwidth partitioning functionality. On many shared platforms, throughput-oriented applications and latency-sensitive applications may be scheduled together — for instance, an interactive application and a streaming application may share the system. While the applications may occupy disjoint cores, they share memory resources such as the last-level cache and memory bandwidth. Therefore, the latency-sensitive application may have unexpected performance slow down due to interferences.

As explained in Section 3, modern hardware often enables cache partitioning and memory bandwidth allocation to control the interference between jobs and improve the quality of service. AMCilk exposes these functionalities to the AMCilk scheduler through an easy-to-use interface allowing the system administrator to manage cores, last-level cache, and memory bandwidth at the same time.

To understand the impact of these functionalities on performance, we run one latency-sensitive application along with a streaming application. The streaming application runs in parallel and repeatedly loads data from memory, modifies it, and stores it back. The latency-sensitive application is an interactive service where clients send requests to the service and the service tries to minimize average flow time (using the DREP scheduler described above in Section 6.1). Since we wish to understand the impact of cache and bandwidth, each job in this latency-sensitive application is a sorting job (since sorting is moderately memory intensive) and the size of jobs vary — 95% of the jobs are short (sorting $500,000$ numbers) and the other 5% are long (sorting $50,000,000$ numbers). We run the latency-sensitive application on cores 2–23 and the streaming application on cores 24–39. In the experiment, the strategy of cache partitioning and band-

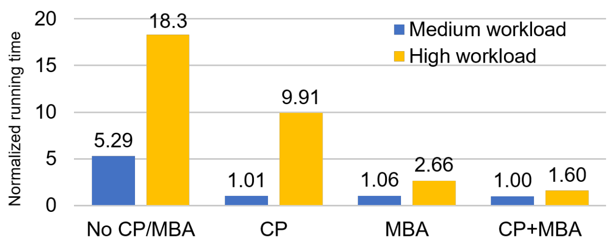


Figure 11: We co-run streaming and latency-sensitive jobs with four settings (horizontal axis): (1) No CP/MBA where neither cache partitioning nor memory bandwidth allocation is set between the two jobs; (2) CP where only cache partitioning is set; (3) MBA where memory bandwidth allocation is set; (4) CP+MBA where both cache partitioning and memory bandwidth allocation are set. We run the four settings with two workloads (shown in legend): (1) Medium workload where the frequency of job requests is medium; (2) High workload where the request frequency is high.

width allocation is simple, since we only want to emphasize the importance of the two functionalities. We allocate a small number of cache columns and a little memory bandwidth to the streaming application while giving a large amount of cache and bandwidth to the latency-sensitive application. The allocation of the cache and memory bandwidth does not change throughout the execution. We compare the average flow time of jobs of the latency-sensitive application between the settings with the cache and memory bandwidth partitioning and the setting without the partitioning.

Figure 11 shows the impact of the streaming application on the average flow time of the interactive application. As a baseline, we ran the interactive application alone (without the streaming application) and use its average flow time to normalize the results of different co-running scenarios. When co-running without any cache or memory bandwidth partitioning, the average flow time increases to 5.29 times for medium load and 18.3 times for high load. Only applying cache partitioning already improves the performance significantly, especially for medium load where the impact of the streaming application virtually disappears. In the setting, we only give 3 cache columns to the streaming application while we give 8 columns to the interactive application. Cache partitioning has minimal impact on its performance since the streaming application itself is insensitive to cache size. For high load, we see further improvement as we apply memory bandwidth allocation. In the setting, we give 10% bandwidth to the streaming application and 90% bandwidth to the interactive application. Finally, we get virtually all of the performance back when we use both cache partitioning and memory bandwidth allocation. Noted that reducing memory bandwidth allocation does have an impact on the streaming application – causing about 150% slowdown (reducing the processing speed from 1855.64 to 724.14 Mflop/sec).

This experiment shows that it is crucial to use cache partitioning and memory bandwidth allocation if we wish to get good performance in multiprogrammed environments. AMCilk allows system administrators to easily access these functionalities using an easy-to-use interface.

6.5. Subscription of runtime information

We now show that the subscription of runtime information is efficiently implemented in AMCilk. Since the overhead incurred by the collection of runtime information is negligible, and the information publishing does not block the critical path of the job execution, we focus on presenting the timing precision to see whether the information in the subscription represent the runtime internals of AMCilk accurately.

We developed a program which has two phases — a high parallelism phase and a low parallelism phase — and the program alternates between the two. The high-parallelism phase keeps all processors busy and almost all the processors are idle during the low parallelism phase. Recall that when a worker runs out of work, it tries to steal work from others. When parallelism is low, workers have a hard time finding work and therefore, they repeatedly steal. Therefore, we expect many steal attempts during the low parallelism phase and very few steal attempts during the high-parallelism phase.

We used this benchmark to measure the accuracy of the information published by the AMCilk runtime system. We set the information to be published

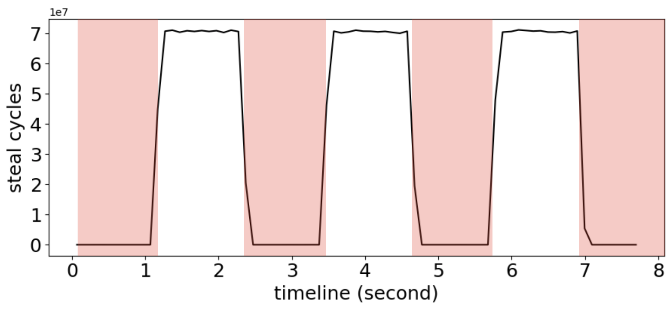


Figure 12: The number of steal cycles on processor 2 in the subscription of runtime information. We run a program which repeatedly switch between high-parallelism phase (red) and low-parallelism phase (white). At 0.1 second intervals, we measure the steal cycles (vertical axis) of processor 2 which is the total time (nanoseconds) the processor 2 spends on work stealing in the previous interval. This information is published to the subscriber and the black line represents the subscriber's view of this information.

every 0.1 seconds. Figure 12 shows the number of steal cycles on processor 2 (as an example), collected by an external process (on the same machine) via the subscription of AMCilk runtime information. The red zone is the time when the program is in the high-parallelism phase. The white zone is the timing when the program is in the low-parallelism phase. The black line denotes the number of steal cycles viewed by the external process via the subscription. We see that when the program enters the low-parallelism phase (white), the number of steal cycles dramatically increases, and the subscriber is able to see this change quite rapidly. Similarly, when the program enters the high-parallelism phase, the number of steal cycles drops to 0 and the subscriber is able to see this change rapidly after it happens. This experiment provides evidence that the subscriber can rapidly get an accurate view of the runtime information allowing it to make appropriate scheduling decisions.

7. Related Work

7.1. Dynamic core reallocation between parallel jobs

The primary feature of AMCilk is the fast and low-overhead core reallocation mechanism between parallel jobs. There has been intensive prior work over a decade ago. Some prior works [25, 26, 27] consider dynamic core reallocation between parallel jobs in threading primitives. In these works, the parallel job is implemented with lightweight threads. AMCilk is different from these works since the parallel jobs running in AMCilk are written in fork-join primitives with language support. As a result, the problem and the design of core reallocation in AMCilk are different from those prior works. Moreover, AMCilk is ready to use since existing legacy written in Cilk language can run in AMCilk without any modification.

Similar to AMCilk, there are prior works [28, 29, 30, 31] that consider dynamic core reallocation between the applications written in fork-join primitives with language support. However, the parallel job in these prior works executes in a work-sharing model, where each worker iteratively takes a chunk of work from a centralized queue and processes it. AMCilk is different from those prior works, where the parallel job executes in the work-stealing model. In the work-stealing model, the work is assigned to workers in a decentralized manner. Thus, the problem and the solution of core reallocation in AMCilk are different from those prior works.

There are prior works [10, 32, 33, 34] that consider the dynamical core reallocation between the applications in the work-stealing model. However, in these works, putting a worker to sleep is either achieved at the steal boundary or lacking in description. In Cilk-AP [35], putting a worker to sleep is achieved at the frame boundary. However, Cilk-AP handles the leftover work of a sleeping worker by work stealing. On the other hand, AMCilk handles the leftover work by mugging, which maintains the number of deques to steal to be equal to the number of cores of a job. Thus, AMCilk keeps the theoretical bound of the work-stealing scheduler [1]. Moreover, none of those systems supports customizable scheduler nor cache and memory bandwidth management. In addition, those systems do not support resource occupancy control or admission control, or subscription of runtime information. On the other hand, AMCilk supports all these functionalities, which makes AMCilk be an efficient runtime system for multiprogrammed parallel applications in shared environments.

7.2. Scheduling multiprogrammed parallel workloads

There is extensive theoretical work on scheduling multiprogrammed parallel workloads in various situations and for different metrics. For example, Edmonds et al. [36] designed a dynamic equipartitioning strategy, which provides a variety of theoretical advantages. For online systems, researchers have considered minimizing average flow time [7, 8, 9, 10], maximum flow time [11, 12], makespan [13] and tail-latency [14, 15, 16]. Various real-time scheduling policies for parallel jobs also require support for multiple jobs running in a single machine [17, 18, 19, 20, 21, 22]. AMCilk is specifically designed to support the above types of scheduling algorithms in an efficient manner.

Several platforms were implemented for various real-world applications, from interactive cloud services [37, 14, 15, 10] to parallel real-time systems [17, 38, 22]. Among them, some platforms can only run the jobs of the specifically modified application program [37, 14, 17]; some create one runtime system for each program and can only support their particular scheduling algorithms [38, 22]; the others use one runtime for multiple jobs, but do not support responsive core reallocation nor the different scheduling algorithms [15, 10]. AMCilk is an efficient platform that meets the requirements of real-world applications and various scheduling algorithms.

In addition, there is intensive prior work on co-scheduling the mix of parallel applications for various performance goals [39, 40, 41, 42, 43, 44, 45, 46], which relies on dynamic core reallocation between the applications and exposure of application runtime. We believe that the fast core reallocation and the support of the shared environment of AMCilk make those co-scheduling designs efficient in implementations.

8. Conclusion

We presented AMCilk, a framework for multiprogrammed parallel workloads based on the Cilk runtime system. AMCilk allows system administrators to customize scheduling policies to support various application scenarios and performance metrics via the low-overhead and responsive core reallocation mechanism and cache and memory bandwidth partitioning. Supporting multiprogrammed workloads efficiently and flexibly is crucial when running large scale systems. While AMCilk is designed for shared memory systems and for a particular language, we believe that the lessons learned in the implementation and performance evaluation of AMCilk are more generally applicable in the design of both small and large-scale systems, such as servers and clouds. The fact that we were able to implement the different applications described in Section 6 indicates that it is possible to design a unified framework that can be easily customized for specific application needs. In addition, our experience with the applications indicates low-overhead and responsive preemption can significantly impact the performance of these applications along with the metrics that these applications care about. On the other hand, running different jobs in the same runtime system potentially causes issues. The most obvious problem is the safety issue when different jobs share the memory, where one job may expose the data to another job. To mitigate this safety issue, a memory allocation mechanism for jobs is highly desirable. In particular, a global variable created by a job should be only visible to the threads within the same job. It would be greatly beneficial to have such constructors when multiprogrammed jobs run on the shared memory platform.

Acknowledgment

This research was supported, in part, by the National Science Foundation (USA) under Grant Numbers CNS–1948457, CCF-1733873 and CCF–1725647.

References

[1] N. S. Arora, R. D. Blumofe, C. G. Plaxton, Thread scheduling for multiprogrammed multiprocessors, in: 10th Annual ACM Symposium on Parallel Algorithms and Architectures, 1998, pp. 119–129.

[2] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, Y. Zhou, Cilk: An efficient multithreaded runtime system, in: Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 1995, pp. 207–216.

[3] M. Frigo, C. E. Leiserson, K. H. Randall, The implementation of the cilk-5 multithreaded language, in: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 1998, pp. 212–223.

[4] Intel Corporation, Intel® Cilk™ Plus Language Extension Specification, Version 1.1, document 324396-002US. Available from `http://cilkplus.org/sites/default/files/open_specifications/Intel_Cilk_plus_lang_spec_2.htm` (2013).

[5] OpenMP: A proposed industry standard API for shared memory programming, OpenMP white paper (Oct. 1997).
URL `http://www.openmp.org/specs/mp-documents/paper/paper.ps`

[6] Intel Corporation, Intel(R) Threading Building Blocks, available from `https://www.threadingbuildingblocks.org/documentation` (2012).

[7] J. Robert, N. Schabanel, Non-clairvoyant scheduling with precedence constraints, in: Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms, SODA '08, 2008, pp. 491–500.

[8] J. Edmonds, K. Pruhs, Scalably scheduling processes with arbitrary speedup curves, ACM Transactions on Algorithms 8 (3) (2012) 28.

[9] K. Agrawal, J. Li, K. Lu, B. Moseley, Scheduling parallel dag jobs online to minimize average flow time, in: Proceedings of the Twenty-seventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), 2016, pp. 176–189.

[10] K. Agrawal, I. A. Lee, J. Li, K. Lu, B. Moseley, Practically efficient scheduler for minimizing average flow time of parallel jobs, in: IEEE Interna-

tional Parallel and Distributed Processing Symposium (IPDPS), 2019, pp. 134–144.

[11] K. Pruhs, J. Robert, N. Schabanel, Minimizing maximum flowtime of jobs with arbitrary parallelizability, in: WAOA, 2010, pp. 237–248.

[12] K. Agrawal, J. Li, K. Lu, B. Moseley, Scheduling parallelizable jobs online to minimize the maximum flow time, in: Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '16, 2016, pp. 195–205.

[13] K. Agrawal, C. E. Leiserson, Y. He, W. J. Hsu, Adaptive work-stealing with parallelism feedback, ACM Transactions on Computer Systems (TOCS) 26 (3) (2008) 1–32.

[14] M. E. Haque, Y. hun Eom, Y. He, S. Elnikety, R. Bianchini, K. S. McKinley, Few-to-many: Incremental parallelism for reducing tail latency in interactive services, in: ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2015, pp. 161–175.

[15] J. Li, K. Agrawal, S. Elnikety, Y. He, I.-T. A. Lee, C. Lu, K. S. McKinley, Work stealing for interactive services to meet target latency, in: Proceedings of the 21st ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '16), 2016, pp. 14:1–14:13.

[16] T. Kaler, Y. He, S. Elnikety, Optimal reissue policies for reducing tail latency, in: Proceedings of the 29th ACM Symposium on Parallelism in Algorithms and Architectures, ACM, 2017, pp. 195–206.

[17] K. Lakshmanan, S. Kato, R. Rajkumar, Scheduling parallel real-time tasks on multi-core processors, in: 31st IEEE Real-Time Systems Symposium (RTSS), 2010, pp. 259–268.

[18] G. Nelissen, V. Berten, J. Goossens, D. Milojevic, Techniques optimizing the number of processors to schedule multi-threaded tasks, in: Proceedings

of the Euromicro Conference on Real-Time Systems (ECRTS), 2012, pp. 321–330.

[19] J. Li, J.-J. Chen, K. Agrawal, C. Lu, C. Gill, A. Saifullah, Analysis of federated and global scheduling for real-time parallel tasks, in: Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS), 2014.

[20] X. Jiang, X. Long, N. Guan, H. Wan, On the decomposition-based global edf scheduling of parallel real-time tasks, in: 2016 IEEE Real-Time Systems Symposium (RTSS), IEEE, 2016, pp. 237–246.

[21] J. Li, D. Ferry, S. Ahuja, K. Agrawal, C. Gill, C. Lu, Mixed-criticality federated scheduling for parallel real-time tasks, in: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2016, pp. 1–12.

[22] J. Orr, C. Gill, K. Agrawal, S. Baruah, C. Cianfarani, P. Ang, C. Wong, Elasticity of workloads and periods of parallel real-time tasks, in: Proceedings of the 26th International Conference on Real-Time Networks and Systems (RTNS), 2018, pp. 61–71.

[23] Z. Wang, C. Xu, K. Agrawal, J. Li, Amcilk: A framework for multiprogrammed parallel workloads, in: 2020 IEEE 27th International Conference on High Performance Computing, Data, and Analytics (HiPC), 2020, pp. 212–222. `doi:10.1109/HiPC50609.2020.00035`.

[24] Intel, User space software for Intel(R) Resource Director Technology., `https://github.com/intel/intel-cmt-cat`.

[25] J. Fried, Z. Ruan, A. Ousterhout, A. Belay, Caladan: Mitigating interference at microsecond timescales, in: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), 2020, pp. 281–297.

[26] A. Ousterhout, J. Fried, J. Behrens, A. Belay, H. Balakrishnan, Shenango: Achieving high cpu efficiency for latency-sensitive datacenter workloads, in:

16th USENIX Symposium on Networked Systems Design and Implementation (NSD 19), 2019, pp. 361–378.

[27] H. Qin, Q. Li, J. Speiser, P. Kraft, J. Ousterhout, Arachne: Core-aware thread management, in: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), 2018, pp. 145–160.

[28] Y. Cho, C. A. C. Guzman, B. Egger, Maximizing system utilization via parallelism management for co-located parallel applications, in: Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, 2018, pp. 1–14.

[29] Y. Cho, S. Oh, B. Egger, Adaptive space-shared scheduling for shared-memory parallel programs, in: Job Scheduling Strategies for Parallel Processing, Springer, 2015, pp. 158–177.

[30] T. Harris, M. Maas, V. J. Marathe, Callisto: Co-scheduling parallel runtime systems, in: Proceedings of the Ninth European Conference on Computer Systems, 2014, pp. 1–14.

[31] J. H. Schönherr, J. Richling, H.-U. Heiss, Dynamic teams in openmp, in: 2010 22nd International Symposium on Computer Architecture and High Performance Computing, IEEE, 2010, pp. 231–237.

[32] J. Li, K. Agrawal, S. Elnikety, Y. He, I.-T. A. Lee, C. Lu, K. S. McKinley, Work stealing for interactive services to meet target latency, in: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2016, pp. 1–13.

[33] Q. Chen, L. Zheng, M. Guo, Adaptive demand-aware work-stealing in multi-programmed multi-core architectures, Concurrency and Computation: Practice and Experience 28 (2) (2016) 455–471.

[34] Y. Cao, H. Sun, D. Qian, W. Wu, Stable adaptive work-stealing for concurrent multi-core runtime systems, in: 2011 IEEE International Conference

on High Performance Computing and Communications, IEEE, 2011, pp. 108–115.

[35] S. Sen, Dynamic processor allocation for adaptively parallel work-stealing jobs, 2004.

[36] J. Edmonds, D. D. Chinn, T. Brecht, X. Deng, Non-clairvoyant multiprocessor scheduling of jobs with changing execution characteristics, Journal of Scheduling 6 (3) (2003) 231–250.

[37] M. Jeon, Y. He, S. Elnikety, A. L. Cox, S. Rixner, Adaptive parallelism for web search, in: ACM European Conference on Computer Systems (EuroSys), 2013, pp. 155–168.

[38] J. Li, S. Dinh, K. Kieselbach, K. Agrawal, C. Gill, C. Lu, Randomized work stealing for large scale soft real-time systems, in: IEEE Real-Time Systems Symposium (RTSS), 2016, pp. 203–214.

[39] N. Kulkarni, G. Gonzalez-Pumariega, A. Khurana, C. A. Shoemaker, C. Delimitrou, D. H. Albonesi, Cuttlesys: Data-driven resource management for interactive services on reconfigurable multicores, in: 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), IEEE, 2020, pp. 650–664.

[40] S. Chen, C. Delimitrou, J. F. Martínez, Parties: Qos-aware resource partitioning for multiple interactive services, in: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, 2019, pp. 107–120.

[41] G. Georgakoudis, H. Vandierendonck, P. Thoman, B. R. D. Supinski, T. Fahringer, D. S. Nikolopoulos, Scalo: Scalability-aware parallelism orchestration for multi-threaded workloads, ACM Transactions on Architecture and Code Optimization (TACO) 14 (4) (2017) 1–25.

[42] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, C. Kozyrakis, Heracles: Improving resource efficiency at scale, in: Proceedings of the 42nd Annual International Symposium on Computer Architecture, 2015, pp. 450–462.

[43] G. Prekas, M. Primorac, A. Belay, C. Kozyrakis, E. Bugnion, Energy proportionality and workload consolidation for latency-critical applications, in: Proceedings of the Sixth ACM symposium on cloud computing, 2015, pp. 342–355.

[44] A. Collins, T. Harris, M. Cole, C. Fensch, Lira: Adaptive contention-aware thread placement for parallel runtime systems, in: Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers, 2015, pp. 1–8.

[45] S. Libutti, G. Massari, P. Bellasi, W. Fornaciari, Exploiting performance counters for energy efficient co-scheduling of mixed workloads on multicore platforms, in: Proceedings of Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures and Design Tools and Architectures for Multicore Embedded Computing Platforms, 2014, pp. 27–32.

[46] M. Bhadauria, S. A. McKee, An approach to resource-aware co-scheduling for cmps, in: Proceedings of the 24th ACM International Conference on Supercomputing, 2010, pp. 189–199.