AMCilk: A Framework for Multiprogrammed Parallel Workloads

Zhe Wang*, Chen Xu*, Kunal Agrawal*, Jing Li†
*Washington University in St. Louis, †New Jersey Institute of Technology
*{zhe.wang, chenxu, kunal}@wustl.edu, †jingli@njit.edu

***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. The primary feature of AMCilk is the low-overhead and responsive preemption 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**

I. 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 [1–4], 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 [7–16] and guaranteeing no deadline misses [17–22]. 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. 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.

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. 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.
- 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.

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. Additionally, AMCilk exposes the resource allocation interface to external systems enabling the control of the resources used by AMCilk.

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 four application scenarios using AMCilk to understand whether AMCilk provides performance improvement to their application-specific criteria. 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 AMCilk provides a performance advantage of between 60 to 70% over the previous implementation, which uses the same scheduling policy — the difference 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 last application demonstrates the importance of cache and memory bandwidth partitioning in multiprogrammed environments.

II. Background

AMCilk is implemented for the Cilk language using a homegrown 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 is 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 [1–3], 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 III.

III. AMCilk Scheduling Framework

This section describes the key implementation details of the AMCilk scheduling framework. In particular, AMCilk uses ***client-server architecture*** (§III-A) 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.

AMCilk scheduling framework provides ***policy-customization interfaces*** (§III-B) that allows 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 ***decentralized AMCilk scheduler*** (§III-C), which is called automatically at pre-defined 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*** (§III-D). 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*** (§III-E). 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.

A. 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 stores. Both request receiver and AMCilk scheduler are nonblocking — they do not wait for a job request to complete before starting on the next one.

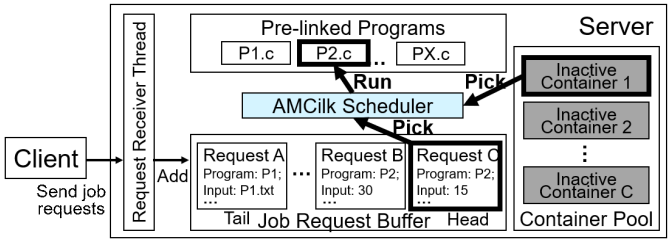

Fig. 1. AMCilk's Client-Server Architecture.

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

B. 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

[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.

Section V, namely (1) DREP; (2) ELASTIC_RT; and (3) PARALLELISM_FB. System administrators 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 pre-defined functions to both get information about the current state of the runtime 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 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. Once she is done, these 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 III-D.

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.

C. Decentralized AMCilk Scheduler

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 2 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).

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 [23] 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 3.

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*** 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

[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.

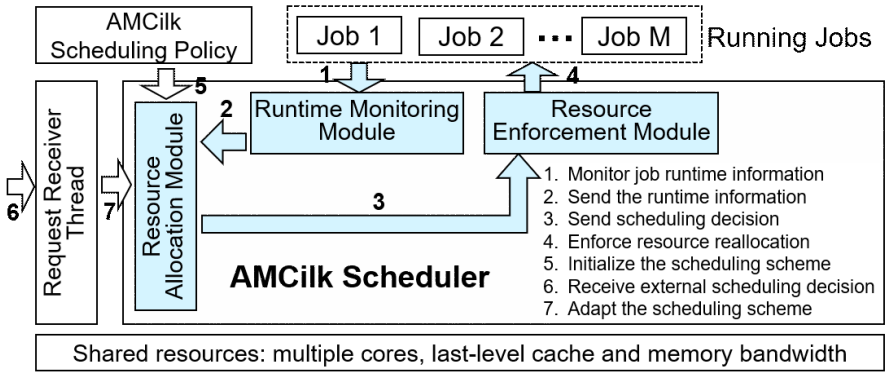

Fig. 2. AMCilk scheduling framework.

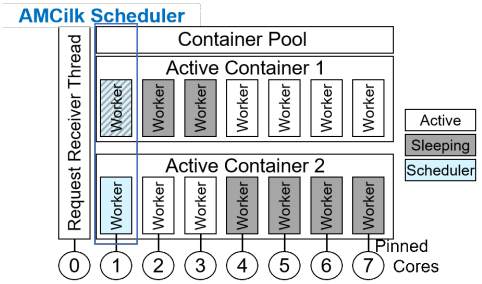

Fig. 3. Runtime snapshot. Container 1 is allocated with 4 cores with 4 active workers, while Container 2 is allocated with 2 cores.

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.

D. 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 previews 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 II, 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 II) 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 III-E, 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. Finally, the overhead is

small since it does not use heavy-duty kernel functions.

E. 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 4.

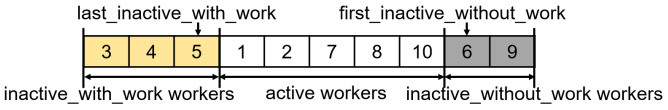

Fig. 4. A job's worker array, storing all its workers sorted in a way that makes it easy for active workers to mug and steal.

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.

IV. 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 40 physical cores (two 2.40GHz Intel Xeon CPUs that support Intel RDT). 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.

A. System Overhead

We first conduct experiments to quantify the time costs of the four core functionalities that AMCilk promises (as discussed in Section III): (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], where requests arrive over time following a Poisson distribution. 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 5.

Evaluation Results. As explained in Section III-C, ***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. 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

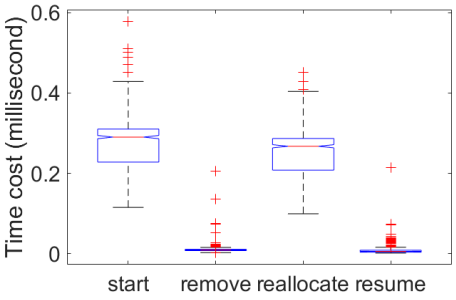

Fig. 5. Time cost of core funcationalities

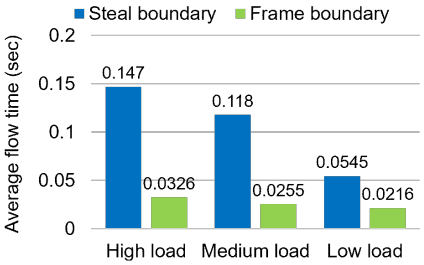

Fig. 6. Average flow time with Bing Search Workload

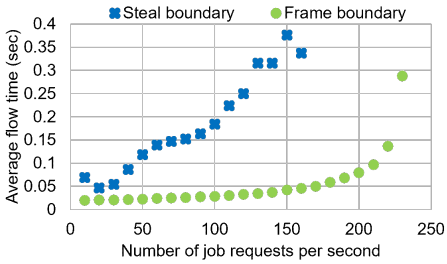

Fig. 7. Average flow time with different request frequency

workers and allocating and (more importantly) initializing the data structures for the closures, frames, and 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. 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), 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. 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 grabbing 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.

B. 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 I, when co-running with the streaming job, the data-intensive job's running time increases by 13.4%. With cache partitioning, the job running time reduces by 2.8%. With memory bandwidth allocation (restricting 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.

V. Case Studies

Multiprogrammed applications are ubiquitous. In this section, we present four concrete examples of multiprogrammed application scenarios with differing needs. We implemented

TABLE I
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

all four scenarios using AMCilk 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?

A. 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 criterion. 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 [7–10]. 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. While the details are not important, when a new job arrives, DREP allocates some of the cores that were previously working on other jobs to this new job and when a job leaves, DREP allocates the cores working on this job to other jobs.

In Agrawal et al.'s implementation [10], preemption only occurs at steal boundaries (as described in Section III). 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[3] 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 generate 100,000 jobs and record the average flow time. Figure 6 shows the results for the bing workload (the finance workload results are even better). The results shows that the frame-boundary implementation reduces the average flow times for 60-70% compared to the steal-boundary implementation. Figure 7 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.

B. 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.[4]

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

[3]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.

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

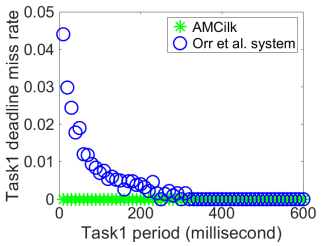

Fig. 8. Elastic scheduling

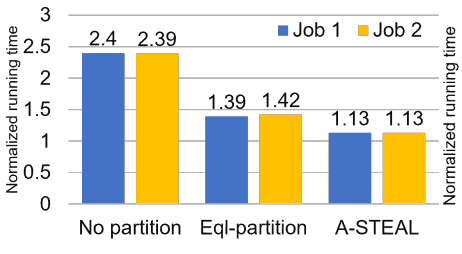

Fig. 9. Adaptive scheduling

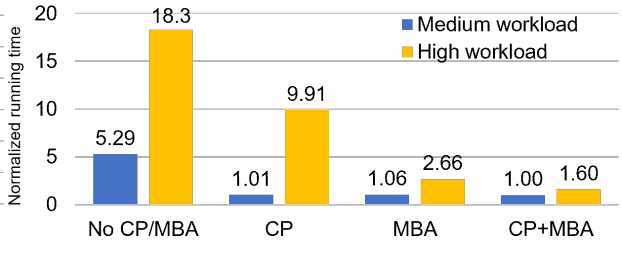

Fig. 10. Co-run streaming and latency-sensitive jobs

experiment with 2 tasks. We vary Task 1's period from 10 to 600 milliseconds, while fixing Task 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 8 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.

C. 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. 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. 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 9, 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.

D. 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 III, 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 V-A). 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 core 2–23 and the streaming application on core 24–39.

Figure 10 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. 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 (where we give 10% of the bandwidth to the streaming 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.

VI. Related Work

Co-scheduling multiprogrammed parallel applications. The primary concern of co-scheduling multiple parallel applications comes from the fact that current resource schedulers have little information about the application's runtime systems. To address this problem, Harris and Maas [24] proposed Callisto — a resource management layer to co-schedule parallel runtime systems. Application runtime systems tell Callisto to run their parallel tasks and Callisto schedules the tasks and runs the tasks via "upcall" to the application code. Qin and Li [25] proposed Arachne, a core-aware thread management system. In the Arachne, each application always knows exactly which cores it has been allocated and it decides how to schedule application threads on cores. AMCilk takes a different approach compared to these two platforms. It runs multiple applications in a single runtime, such that any worker has a comprehensive view of the entire system runtime, which enables highly flexible scheduling policies.

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. [26] designed a dynamic equipartitioning strategy, which provides a variety of theoretical advantages. For online systems, researchers have considered minimizing average flow time [7–10], maximum flow time [11], [12], makespan [13] and tail-latency [14–16]. Various real-time scheduling policies for parallel jobs also require support for multiple jobs running in a single machine [17–22]. 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 [10], [14], [15], [27] to parallel real-time systems [17], [22], [28]. Among them, some platforms can only run the jobs of the specifically modified application program [14], [17], [27]; some create one runtime system for each program and can only support their particular scheduling algorithms [22], [28]; the others use one runtime for multiple jobs, but do not support responsive preemption nor the different scheduling algorithms [10], [15]. AMCilk is an efficient platform that meets the requirements of real-world applications and various scheduling algorithms.

VII. 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 V 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 4 applications indicates low-overhead and responsive preemption can significantly impact the performance of these applications along with the metrics that these applications care about.

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

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

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

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

[8] J. Edmonds and K. Pruhs, "Scalably scheduling processes with arbitrary speedup curves," *ACM Transactions on Algorithms*, vol. 8, no. 3, p. 28, 2012.

[9] K. Agrawal, J. Li, K. Lu, and 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, and B. Moseley, "Practically efficient scheduler for minimizing average flow time of parallel jobs," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 134–144.

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

[12] K. Agrawal, J. Li, K. Lu, and 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*, ser. SPAA '16, 2016, pp. 195–205.

[13] K. Agrawal, C. E. Leiserson, Y. He, and W. J. Hsu, "Adaptive work-stealing with parallelism feedback," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 3, pp. 1–32, 2008.

[14] M. E. Haque, Y. hun Eom, Y. He, S. Elnikety, R. Bianchini, and 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, and 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, and 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, and 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, and 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, and 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, and 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, and 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, and 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] Intel, "User space software for Intel(R) Resource Director Technology." https://github.com/intel/intel-cmt-cat.

[24] T. Harris, M. Maas, and V. J. Marathe, "Callisto: Co-scheduling parallel runtime systems," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2592798.2592807

[25] H. Qin, Q. Li, J. Speiser, P. Kraft, and J. Ousterhout, "Arachne: Core-aware thread management," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 145–160. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/qin

[26] J. Edmonds, D. D. Chinn, T. Brecht, and X. Deng, "Non-clairvoyant multiprocessor scheduling of jobs with changing execution characteristics," *Journal of Scheduling*, vol. 6, no. 3, pp. 231–250, 2003.

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

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