Dynamic Processor Allocation for Adaptively Parallel Work-Stealing Jobs

by

Siddhartha Sen

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2004

Author .....
Department of Electrical Engineering and Computer Science
September 10, 2004

Certified by .....
Charles E. Leiserson
Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Dynamic Processor Allocation for Adaptively Parallel Work-Stealing Jobs

by

Siddhartha Sen

Submitted to the Department of Electrical Engineering and Computer Science
on September 10, 2004, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis addresses the problem of scheduling multiple, concurrent, adaptively parallel jobs on a multiprogrammed shared-memory multiprocessor. Adaptively parallel jobs are jobs for which the number of processors that can be used without waste varies during execution. We focus on the specific case of parallel jobs that are scheduled using a randomized work-stealing algorithm, as is used in the Cilk multithreaded language.

We begin by developing a theoretical model for two-level scheduling systems, or those in which the operating system allocates processors to jobs, and the jobs schedule their threads on the processors. To analyze the performance of a job scheduling algorithm, we model the operating system as an adversary. We show that a greedy scheduler achieves an execution time that is within a factor of 2 of optimal under these conditions. Guided by our model, we present a randomized work-stealing algorithm for adaptively parallel jobs, algorithm WSAP, which takes a unique approach to estimating the processor desire of a job. We show that attempts to directly measure a job's instantaneous parallelism are inherently misleading. We also describe a dynamic processor-allocation algorithm, algorithm DP, that allocates processors to jobs in a fair and efficient way. Using these two algorithms, we present the design and implementation of Cilk-AP, a two-level scheduling system for adaptively parallel work-stealing jobs. Cilk-AP is implemented by extending the runtime system of Cilk.

We tested the Cilk-AP system on a shared-memory symmetric multiprocessor (SMP) with 16 processors. Our experiments show that, relative to the original Cilk system, Cilk-AP incurs negligible overhead and provides up to 37% improvement in throughput and 30% improvement in response time in typical multiprogramming scenarios.

This thesis represents joint work with Charles Leiserson and Kunal Agrawal of the Supercomputing Technologies Group at MIT's Computer Science and Artificial Intelligence Laboratory.

Thesis Supervisor: Charles E. Leiserson
Title: Professor

Acknowledgments

I would like to thank Prof. Charles Leiserson for his infinite energy, intelligence, wisdom, and guidance throughout the course of this thesis. Charles is one of very few professors I have met who is just as human and real as he is smart and accomplished. I would also like to thank Dr. Bradley Kuszmaul for his help with systems, and Prof. Michael Bender for many useful discussions on the theoretical aspects of this thesis.

I would like to thank Kunal Agrawal for working with me on all aspects of this project. I would also like to thank the other members of the Supercomputing Technologies Group at MIT: Elizabeth Basha, John Danaher, Jeremy Fineman, Zardosht Kasheff, Angelina Lee, Sean Lie, Tim Olsen, and Jim Sukha. All have contributed in some way or another to this thesis. Special thanks to Tim for helping me get everything together at the last minute.

I would like to thank Abhinav Kumar for his help with mathematical proofs related to our algorithms. Abhinav spent days trying to break a conjecture we initially thought to be true, and eventually managed to break it.

I would like to thank my family for their endless support and love. They have been there for me at every turn of my life, and have tolerated every one of my vices with remarkable grace.

Finally, I would like to thank Faye McNeill for all of her help, support, and love, especially during these past two weeks. She is the reason I was given a heart and the ability to love, and she will stand by my side forever. That's a promise.

Contents

List of Figures

Chapter 1

Introduction

An ***adaptively parallel job*** is a job for which the number of processors that can be used without waste varies during execution. This number represents the instantaneous parallelism, or processor desire, of the job. In this thesis, we investigate the problem of scheduling multiple, concurrent, adaptively parallel jobs on a multiprogrammed shared-memory multiprocessor. We focus on the specific case of parallel jobs that schedule their computation on a given set of processors using a randomized work-stealing algorithm, as is used in the Cilk multithreaded language [5, 19, 47].

The problem of scheduling parallel jobs on multiprogrammed parallel systems is two-fold: first, the processors are allocated to the competing jobs, and second, a thread from a job is chosen to run on each of the job's allocated processors [17]. In ***single-level scheduling***, the decision of where to allocate a processor is combined with the decision of which thread to run on it; in other words, the operating system schedules all threads of all jobs. In ***two-level scheduling***, the two issues are decoupled: the operating system is responsible for allocating processors to the jobs (the first level of scheduling), and the jobs are responsible for scheduling their threads on those processors (the second level of scheduling). Since processors are allocated to jobs, sharing is done using ***space slicing***, where the processors of the system are partitioned statically or dynamically among the different jobs. In contrast, ***time slicing*** shares processors by rotating them from one job to another during each time quantum and is more common in single-level scheduling.

We have developed a theoretical basis for analyzing the performance of two-level schedulers. We model the interaction between the first and second levels of scheduling by playing a game between the operating system and the job scheduler, in which the operating system acts as an adversary. A similar approach is taken by Arora et. al. [2] to model multiprogrammed environments for their thread scheduler. We define an adaptively parallel job using the graph-theoretic model of multithreaded computation developed by Blumofe and Leiserson [7]. We extend the model's definition, however, to support the idea of adaptive parallelism when executing a multithreaded computation. In particular, the model can handle a scenario where the operating system (adversary) changes the number of processors allocated to a job at the beginning of every time step. We show that it is still possible for a job scheduler using a greedy algorithm under these conditions to achieve an execution time that is within a factor

of 2 of optimal.

Using our theoretical model, we have developed a randomized work-stealing algorithm for adaptively parallel jobs, algorithm WSAP (for "work stealing, adaptively parallel"), that can handle dynamically changing processor allocations; WSAP is an extension of algorithm WS presented in [7]. We also describe a dynamic processor-allocation algorithm, algorithm DP (for "dynamic partitioning"), which allocates processors to jobs both fairly and efficiently by responding to changes in the processor desires of jobs during runtime. Using these two algorithms, we have designed and implemented a two-level scheduling system for adaptively parallel work-stealing jobs running on a multiprogrammed shared-memory system. In the first level of scheduling, we use algorithm DP to allocate processors to the running jobs, based on their current processor desires. In the second level of scheduling, we use algorithm WSAP to schedule the computation of a single job and report its current processor desire to the first-level scheduler.

The manner in which a job's current processor desire is estimated is an interesting problem on its own. In particular, we show that measuring the instantaneous parallelism of a job can grossly underestimate the actual parallelism in the job. We propose a strategy that proactively explores the parallelism of a job and uses feedback from work-stealing statistics to tune the estimated desire.

We have implemented our two-level scheduler by extending the runtime system of the Cilk multithreaded language. We call our system Cilk-AP. The original Cilk scheduler implements algorithm WS [7], which assumes that the number of processors allocated to a job remains fixed throughout its execution. We modified the scheduler to implement algorithm WSAP, which handles dynamically changing processor allocations, and we added a user-level extension that implements the processor-allocation algorithm DP. Cilk-AP uses a technique called process control [48] to coordinate the reallocation of processors between the first and second levels of scheduling. This technique is different from the technique used in [2], because it ensures that the number of virtual processors used by a job always matches the number of physical processors assigned to it. Empirical results demonstrate that Cilk-AP has low overhead and improved performance over the original Cilk system in a variety of situations. These results also suggest that algorithm WSAP runs in asymptotically optimal time, although we have not yet verified this conjecture theoretically.

The remainder of this thesis is organized as follows. In Chapter 3, we develop a theoretical model for adaptively parallel computation, using the two-level scheduling approach. Chapter 4 presents our randomized work-stealing algorithm for adaptively parallel jobs, algorithm WSAP, and Chapter 5 presents our dynamic processor-allocation algorithm, algorithm DP. In Chapter 6, we discuss the design and implementation of our two-level scheduling system, Cilk-AP; we present empirical results for Cilk-AP in Chapter 7. Chapter 8 discusses previous work and we conclude in Chapter 9.

Chapter 2

High-Level Policy Decisions

In this chapter, we justify some of the major policy decisions in our approach to scheduling adaptively parallel jobs. In particular, we discuss the advantages of

1) two-level scheduling over single-level scheduling,
2) dynamic processor allocation over static allocation , and
3) coordinated processor reallocation over uncoordinated processor reallocation.

When choosing these policies, we focus on their applicability to shared-memory systems that use the "workpile-of-tasks" programming model, in which a computation is represented as a workpile of tasks that are executed by a variable number of worker threads [17]. The model for adaptively parallel computation that we present in Chapter 3 is a variant of this model.

Two-Level Scheduling

The first policy decision we consider is the choice between single-level and two-level scheduling, the two basic approaches to scheduling in multiprogrammed parallel systems defined in Chapter 1.

In single-level scheduling, the operating system is responsible for scheduling all job threads directly onto the processors. The problem with this approach is that it incurs high operating-system overhead and may not be responsive to application needs [17]. Since many scheduling decisions are a result of synchronization conditions among the application's threads, paying the operating system overhead at every synchronization point is expensive. The cost is especially high in fine-grained applications, where such interactions between threads are numerous and frequent. Furthermore, the operating system is unable to optimize the scheduling, because it lacks information about the application's characteristics and synchronization patterns.

The solution to this problem is to use two-level scheduling, where the operating system is only responsible for allocating processors to the competing jobs, and the applications themselves perform the fine-grain scheduling of threads onto allocated processors in a way that satisfies synchronization constraints. The internal level of scheduling allows the application to have more threads than allocated processors.

It also makes it possible to create systems where the allocation changes at runtime and applications are expected to reschedule their threads accordingly. This approach is well suited to the workpile-of-tasks programming model, where the tasks can be executed by a variable number of worker threads.

While two-level scheduling is not universally accepted for all types of multiprogrammed parallel systems, it is perfectly suitable for shared-memory systems using the workpile-of-tasks programming model [17], which are the focus of this thesis. The presence of shared memory makes it possible for tasks to be executed by any worker thread on any processor. In fact, if the shared memory is centrally located, memory allocation (e.g. to implement the shared workpile) is completely decoupled from the allocation of processors. Previous work has also shown that single-level schedulers tend to work poorly on shared-memory systems [22,38,50,51].

Dynamic Processor Allocation

The first level of a two-level scheduling system allocates processors to jobs using either a static or dynamic allocation policy. In a ***static*** policy, the number of processors assigned to a job remains fixed throughout the job's execution. In a ***dynamic*** policy, the number of allocated processors may change during runtime. The current Cilk scheduler can only be used with a static processor-allocation policy, since it assumes that a job's processor allocation does not change during runtime. The Cilk-AP scheduler presented in Chapter 6, on the other hand, copes with dynamically changing allocations, and so it is suitable for use with a dynamic policy.

The disadvantage of a static processor-allocation policy is that it leads to fragmentation of processor resources and it compromises fairness. Since job allocations are fixed during runtime, new arrivals are forced to wait in a queue if sufficient processors are not available. Various strategies to reduce queuing have been proposed [16,42,43,49], but all of these strategies essentially reserve processors for future arrivals, thus wasting system resources and limiting achievable utilization. In addition, static policies are unable to adapt to changes in processor requirements exhibited by jobs during runtime.

Unlike static policies, dynamic processor-allocation policies allow the system to respond to load changes, whether they are caused by the arrival of new jobs, the departure of completed jobs, or changes in the parallelism of running jobs—the last case is of particular importance to us because of our study of adaptively parallel jobs. In all three cases, a dynamic policy is able to redistribute processor resources to accommodate the change in load.

To see how this response compares to that of a static policy, consider the example scenarios presented in Figure 2-1. In scenario (a), jobs A and B start running at time $T = 0$ with $P/2$ processors each, where P is the number of processors in the system, and a third job requesting $P/2$ processors arrives at time $T = t$. In the static case, all processors of the system are busy, so job C is forced to wait until either A or B finishes; this policy compromises fairness and suffers from reduced throughput. The dynamic policy, on the other hand, is able to redistribute the processors of the

system at time t and give each job a fair share of $P/3$ processors, thus achieving better throughput. In scenario (b), only jobs A and B are running in the system, but they exhibit complementary parallelism profiles (indicated by the dashed line). In the static case, no changes can be made to the processor allocations once the jobs have started, so there are periods during which some of the processors are poorly utilized. This problem does not occur in the dynamic case, because the policy is able to redistribute underutilized processors from job A to B and vice versa. The response times of both jobs are also better because the processors are used more efficiently.

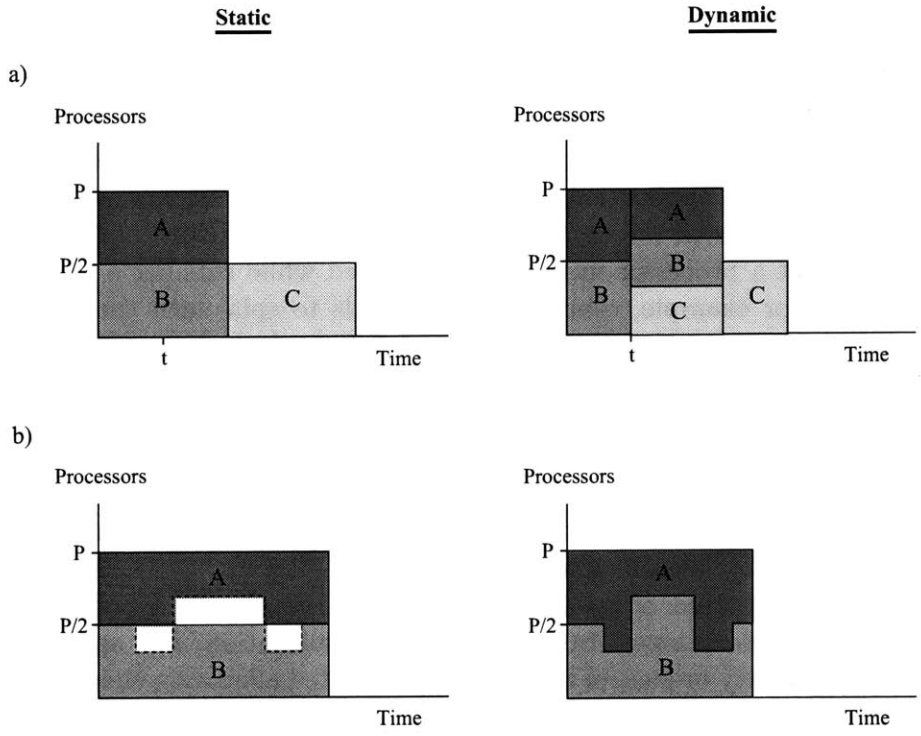


Figure 2-1: Theoretical performance of static vs. dynamic processor-allocation policies on a system with P processors. In scenario a), jobs A and B are running and job C enters the system at time t; in scenario b), jobs A and B are running and exhibit complementary parallelism profiles.

Dynamic processor-allocation policies have been studied and implemented extensively in the past [8, 12, 15, 22, 27, 29, 30, 32, 33, 35, 37, 39–41, 44–46, 48, 52]. Most of this work, however, assumes that the instantaneous parallelism of the jobs is known and used by the scheduling system when making its decisions. In practice, this information is not easily discernible, and in some cases can be inherently misleading. (We show an example of such a case and describe our solution to the problem in Chapter 6.) In particular, at present not only does the Cilk system expect the user to manually enter the parallelism of a job before it begins, but this parallelism remains fixed throughout the job's execution, making the system only suitable for use with static processor allocation policies.

The primary complaint against dynamic processor-allocation policies is that they incur more system overhead than static policies due to the frequent reallocations. This overhead, however, is significantly reduced in shared-memory systems that use the workpile-of-tasks programming model, because the presence of shared memory makes it possible for tasks to be executed by any worker thread on any processor. Moreover, the reallocation overhead also depends on how changes to job allocations are coordinated between the first and second levels of scheduling.

Coordinated Reallocations

An important policy decision for two-level scheduling systems that employ a dynamic processor-allocation policy is the manner in which reallocations are coordinated. In ***uncoordinated*** reallocations, the operating system moves processors without interacting with the application, while in ***coordinated*** reallocations, processors are moved in concert with the application [33]. The problem with uncoordinated reallocations is that a processor might get preempted while running a thread that holds a spin lock, for example, causing other threads to spin until the critical thread is resumed. In fact, the thread need not be holding a lock, but may be critical to the application for another reason, causing other threads to block or spin at the next synchronization point. This waste of processor resources can significantly deteriorate both application and system performance [17, 33]. While several locking protocols have been proposed to alleviate the problem—for example, "two-phase" or "spin-then-block" locking [24, 28]—they neither eliminate the problem nor target threads that are critical for reasons unrelated to locks.

To avoid this kind of reallocation overhead, we choose a policy that performs coordinated reallocations. In a coordinated reallocation, the application is given some responsibility or control to effect the requested allocation change, or at the very least, it receives notifications from the operating system when important scheduling events occur. Since the application knows which of its threads are critical at any given time, it can avoid preempting the corresponding processors in the event that its allocation needs to be reduced. For example, an application that follows the workpile of tasks programming model can avoid preempting a processor whose thread is in the middle of executing a given task.

In general, coordinated reallocations allow us to flexibly distribute reallocation responsibilities between the application and the operating system. The more responsibility that is given to the application, however, the more the operating system needs to trust that it will respond to allocation changes in an honest manner. Several mechanisms have been proposed to implement coordinated reallocation policies, such as scheduler activations [1], first-class user threads [31], and process control [48]. The Cilk-AP system uses the process control mechanism, which we discuss in Section 6.2.

Chapter 3

A Theoretical Model for Scheduling Adaptively Parallel Jobs

In this chapter, we develop a theoretical model for adaptively parallel jobs and their execution schedules. We assume a two-level scheduling environment in which processors are allocated to jobs in the first level using a dynamic policy, as discussed in Chapter 2. We begin by modeling an adaptively parallel job as a "fully strict" (well-structured) multithreaded computation with a given execution schedule, where the number of processors allocated to the job may change at each step of the execution. Then, we model the interaction between the first and second levels of scheduling by playing a game between the operating system and the job scheduler, in which the operating system acts as an adversary; this approach is also used in [2]. We end with an analysis of a greedy job scheduler in this adversarial model, showing that greedy schedules achieve an execution time which is within a factor of 2 of optimal.

3.1 A Model for Adaptively Parallel Jobs

In this section, we present a model for adaptively parallel jobs based on the graph-theoretic model of multithreaded computation described in [7]. We model an adaptively parallel job as a fully strict multithreaded computation with an adaptive execution schedule. We define each of these terms in turn.

Our definition of a multithreaded computation is identical to the definition provided in [7]. To summarize, a multithreaded computation consists of a set of threads that are connected to each other via spawn or data-dependency edges. A ***thread*** is defined as a sequential ordering of unit-size tasks, where each task takes one unit of time to execute on a processor, and tasks in the same thread are connected by ***continue*** edges that specify the sequential ordering of the thread. Figure 3-1 shows an example of a multithreaded computation. During its execution, a thread (which we call the "parent") may create, or ***spawn***, other threads (called "children"). Each spawn is represented by an edge, called the ***spawn*** edge, which connects the task in

the parent thread that performed the spawn operation to the first task of the resulting child thread. A spawn is similar in semantics to a subroutine call, except that the parent thread is allowed to operate concurrently with the child thread.

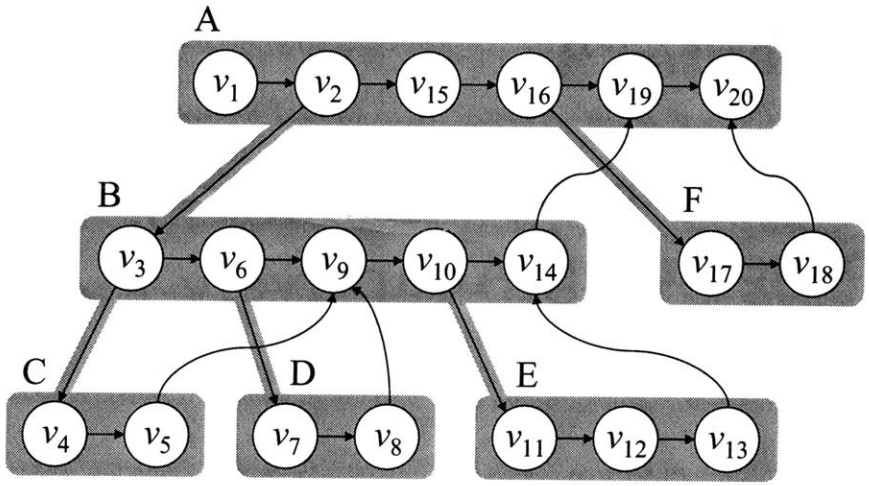


Figure 3-1: A fully strict multithreaded computation. The shaded blocks represent threads, the circles represent tasks, and the horizontal edges within each thread are continue edges. Threads are connected to each other by spawn edges, shown as downward-pointing, shaded edges, and dependency edges, shown as curved edges. This computation contains 20 tasks $v_1, v_2, \ldots, v_{20}$ organized into six threads.

In addition to spawn edges, a multithreaded computation may also contain dependency edges between threads, modeling both data and control dependencies. Dependency edges allow threads to synchronize with each other and must be satisfied before a thread can continue executing. For example, task v_9 in Figure 3-1 cannot execute before tasks v_5 and v_8 have completed because of the dependency edges from v_5 to v_9 and v_8 to v_9. If the execution of thread B reaches v_9 and these dependencies have not been resolved, thread B ***stalls***; the execution of v_5 and v_8 ***enables*** thread B, making it ***ready*** for execution.

In general, a multithreaded computation can be viewed as a directed acyclic graph (dag) of tasks connected by continue, spawn, and dependency edges. Since it can be impossible to schedule multithreaded computations with arbitrary dependencies efficiently [6], we focus on a subclass of multithreaded computations, called ***fully strict*** computations, in which all dependency edges from a thread go to its parent. We say the computation is "well-structured" in the sense that all dependencies from a subcomputation emanate from the subcomputation's root thread. Any multithreaded computation that can be executed in a depth-first manner on one processor can be made fully strict by altering the dependency structure—possibly at some cost in lost parallelism—but not affecting the semantics of the computation [4, 7]. The computation depicted in Figure 3-1 is fully strict.

An ***execution schedule*** for a multithreaded computation determines which pro-

cessors of a parallel computer execute which tasks at each step [7]. A processor may execute at most one task during any step of the execution schedule. In addition, an execution schedule must obey the constraints delineated by the continue, spawn, and dependency edges of the computation dag. In particular, no processor may execute a task until all of the task's predecessors in the dag have been executed, i.e., the task is ***ready***. The execution schedules studied in [7] assume that the number of processors available to execute a computation remains fixed over time. We remove this assumption in our model and introduce the notion of an ***adaptive*** execution schedule, which is identical to an execution schedule except that the number of available processors is allowed to change (instantaneously) from one step of the execution to the next.

We now have all the tools necessary to model an adaptively parallel job. An adaptively parallel job is a fully strict multithreaded computation with a given adaptive execution schedule. The execution of an adaptively parallel job is controlled by two entities: the operating system, which controls the number of processors allocated to the job during each step of its execution, and the job scheduler, which generates an adaptive execution schedule subject to the allocations dictated by the operating system. These entities correspond to the first and second levels of a two-level scheduling system. In order to analyze the execution time of an adaptively parallel job, we need to model the roles of these two levels, as well as the interaction between them. We present this model in Section 3.2.

3.2 An Adversarial Model for Two-Level Schedulers

In this section, we present a model for two-level scheduling systems based on the model for adaptively parallel jobs from Section 3.1. We describe a game played between the operating system and the job scheduler to execute a job, in which the operating system acts as an adversary. Using this adversarial model, we show how to quantify the execution time of an adaptively parallel job.

There are three main entities in a two-level scheduling system: the operating system (representing the first level of scheduling), the job scheduler (representing the second level of scheduling), and the job itself. The execution of an adaptively parallel job is controlled by both the operating system and the job scheduler. We model the interaction between the first and second levels of scheduling by playing a game between these two entities. We assume that there is only one job in the system initially (extending our model to handle multiple, concurrent jobs is trivially shown at the end of this section). The rules of the game are simple. During each time step t of a job's execution, three operations occur:

1. The operating system determines the number P_t of processors to allocate to the job for step t.
2. The job scheduler assigns tasks in the computation that are ready at time t to one or more of the P_t processors (i.e., it generates one step of the adaptive execution schedule).

3. Each processor that is assigned a task finishes executing the task by the end of step t. (Recall that all tasks in the computational dag are unit-sized.)

We refer to the players in this game as OS and JS, for the operating system and the job scheduler, respectively. The role of OS is to act as an adversary, similar to the role of the operating system in [2]. The role of JS, on the other hand, is to schedule the job as efficiently as possible given the allocations provided by OS, minimizing the job's execution time. There are no restrictions on the policy used by OS to determine the job's allocation during each time step.

To quantify the execution time of the job, we first define some terms. Let P_t be the number of processors allocated to the job during step t, where $t \in S = \{0, 1, \ldots, T\}$. T is the execution time of the job on an ideal machine with no scheduling overhead. In general, we make this assumption for all theoretical execution times in our model. We define the ***critical path length*** of a job to be the length of the longest directed path in the computational dag. We denote this length by T_∞, since with an infinite number of processors, each task along the critical path must still be executed serially. We define the ***work*** of a job to be the total number of tasks in the dag, denoted by T_1, since a single processor can only execute one task during each step. We want to bound the execution time T of the job using these two quantities, similar to the bounds presented in [7].

If the processor allocation of a job remains fixed during its execution (as in a static allocation policy), then our situation reduces to the situation studied in [7]. In particular, if $P_t = P$ for all $t \in S$, then the following bounds can be shown for T: $T \geq T_\infty$ and $T \geq T_1/P$. The first bound holds because any P-processor execution of the computation must execute the critical path. To show the second bound, we note that the P processors can execute at most P tasks per time step, and since the computation has a total of T_1 tasks, the execution time is at least T_1/P. In our model, however, the processor allocation policy used by OS need not be static; that is, the values of P_t may change from one time step to the next. While it still follows that $T \geq T_\infty$ (for the same reason given above), we cannot make the statement that $T \geq T_1/P$, because P is not a fixed number in our case. We can try to prove a similar bound, however, based on the average value of P_t, which we denote by $\overline{P} = (1/T) \sum_{t \in S} P_t$. The problem with using $\overline{P}$ as defined is that, if OS allocates an infinite number of processors to the job during one or more time steps, the value of $\overline{P}$ becomes infinite, yielding a trivial bound of $T \geq T_1/\infty = 0$. To solve this problem, we modify our definition of $\overline{P}$ to cope with infinite allocations. In particular, given the allocations P_t of the job, $t \in S$, let $P_{t_1}, P_{t_2}, \ldots, P_{t_{T_\infty}}$ be the T_∞ largest allocations received by the job during its execution, and let $S' = \{t_1, t_2, \ldots, t_{T_\infty}\}$ be the set of time steps corresponding to those allocations. Notice that the elements of S' are distinct and $S' \subseteq S$. Also, let R_t be the number of ready tasks in the computational dag at time step t. Then, we define $\overline{P}$ as follows:

Definition 1 *The **average effective processor allocation**, $\overline{P}$, is the average of*

all P_t, $t \in S$, with each P_{t_i} replaced by $\min(P_{t_i}, R_{t_i})$ for all $t_i \in S'$. In other words,

$$\overline{P} = \frac{1}{T}\left(\sum_{t \in S-S'} P_t + \sum_{t \in S'} \min(P_t, R_t)\right). \tag{3.1}$$

The modified definition of $\overline{P}$ replaces values of P_t that are infinite with the corresponding values of R_t. This substitution allows us to calculate $\overline{P}$ in the face of adversarial behavior by OS. It also makes intuitive sense, because R_t represents the maximum number of tasks that JS can schedule in a particular step, which is also the maximum number of processors that can be used in that step. Without the modification to $\overline{P}$'s definition, OS could apply the following "winning" strategy to every possible job: allocate an infinite number of processors to the job in the first time step, and then allocate 1 processor during each succeeding step. In this scenario, $\overline{P} = \infty$ and the best execution time JS can achieve is $T = T_1$, since only one task is ready during the first time step and at most one task can be executed during each succeeding step. Thus, the execution time on an infinite number of processors is no better than the serial execution time (when $\overline{P} = 1$).

Using Equation (3.1), we can now prove a lower bound on execution time based on the amount of work in the computation.

Lemma 2 *For any adaptively parallel job with work T_1 and average effective processor allocation $\overline{P}$, the execution time T of the job satisfies the constraint $T \geq T_1/\overline{P}$.*

Proof. During each step t of the job's execution, at most P_t tasks can be executed, since there are only P_t processors available to the job and each processor executes at most 1 task per time step. The sum of the P_t's therefore serves as an upper bound for the amount of work in the job:

$$T_1 \leq \sum_{t \in S} P_t \,. \tag{3.2}$$

Let $P_{t_1}, P_{t_2}, \ldots, P_{t_{T_\infty}}$ be the T_∞ largest allocations received by the job during its execution, as before. If we replace each P_{t_i} with $\min(P_{t_i}, R_{t_i})$, then Inequality (3.2) still holds, because R_t is also an upper limit on the number of tasks that can be executed during step t. Thus, we restate the bound as follows:

$$\begin{aligned} T_1 &\leq \sum_{t \in S-S'} P_t + \sum_{t \in S'} \min(P_t, R_t) \\ &\leq T\left(\frac{1}{T}\right)\left(\sum_{t \in S-S'} P_t + \sum_{t \in S'} \min(P_t, R_t)\right) \\ &\leq T\overline{P} \,. \end{aligned}$$

□

The bounds shown in this section hold even if multiple adaptively parallel jobs are running in the system, each with its own instance of JS. From the point of view of

the JS's, OS is an adversary, and so no expectations or constraints are placed on the allocations it provides to their jobs. Consequently, each JS need not be aware of how many other JS's are in the system or what allocations they receive from OS, which makes sense since the JS has no control over either of these factors. All a JS must know in order to perform its function is the allocation of its own job during each time step (regardless of how this allocation is determined).

3.3 Analysis of Greedy Schedules

For the case of static processor-allocation policies, early work by Graham [20, 21] and Brent [10] shows that there exist P-processor execution schedules that satisfy $T_P \leq T_1/P + T_\infty$, where T_P is the minimum execution time over all P-processor execution schedules. The greedy-scheduling theorem in [6] extends this result by proving the same bound on T_P for greedy schedules, or schedules that attempt to do as much work as possible during every step. In this section, we prove a similar bound for greedy schedules for the case of dynamic processor-allocation policies, where the allocation of a job may change during each step of its execution.

We define an ***adaptive greedy schedule*** to be an adaptive execution schedule in which, during each step t of a job's execution, if at least P_t tasks are ready, then P_t tasks execute, and if fewer than P_t tasks are ready, then all execute. In the scheduling game described in Section 3.2, the adaptive greedy schedule would be generated by JS in response to the allocations P_t provided by OS. We now state the Graham-Brent bound for adaptive greedy schedules:

Theorem 3 (The adaptive greedy-scheduling theorem) *For any adaptively parallel job with work T_1, critical path length T_∞, and average effective processor allocation $\overline{P}$, any adaptive greedy schedule achieves $T \leq T_1/\overline{P} + T_\infty$, where T is the execution time of the job.*

Proof. We classify each step t of a job's execution as one of two types: in a ***complete*** step, there are at least P_t tasks that are ready, so a greedy schedule selects any P_t of them to execute. An ***incomplete step*** has strictly less than P_t ready tasks, and so a greedy schedule executes them all. Since each step is either complete or incomplete, we can bound the time used by an adaptive greedy schedule by bounding the number of complete and incomplete steps.

Consider the complete steps first. If every step $t \in S - S'$ of the job's execution is complete, then exactly P_t tasks are executed during each of these steps by a greedy schedule. Also, since $\min(P_t, R_t) \leq R_t$ for all $t \in S'$, no more than R_t tasks are executed during the remaining steps. Since no more than T_1 tasks can be executed

by the greedy schedule, we have that

$$\begin{aligned} T_1 &\geq \sum_{t \in S - S'} P_t + \sum_{t \in S'} \min(P_t, R_t) \\ &\geq T \cdot \frac{1}{T} \left(\sum_{t \in S} P_t + \sum_{t \in S'} \min(P_t, R_t) \right) \\ &\geq T\overline{P} \,, \end{aligned}$$

from which it follows that $T \leq T_1/\overline{P}$. Thus, the maximum number of complete steps is $T_1/\overline{P}$. Now, consider the number of incomplete steps. Let G denote the computational dag of the job, and let G_t be the subgraph of G that remains at the beginning of step t, where $G_0 = G$. During each step t, all tasks in G_t with in-degree 0 (i.e, tasks that have no unexecuted predecessors) are ready to be executed. If we assume that every step is incomplete, then all of these tasks are executed by a greedy schedule during step t. Since the longest path in G_t starts with a task that has in-degree 0, the length of the longest path in G_{t+1} must be one less than the length of the longest path in G_t. In other words, every incomplete step reduces the length of a longest path in G by one unit; since this length is precisely the critical-path length, there can be at most T_∞ such steps. For example, Figure 3-2 illustrates the progression $\langle G_0, G_1, \ldots, G_T \rangle$ of dags resulting from a greedy schedule of the computation in Figure 3-1, where every step is incomplete; the length of the critical path (shown in bold) reduces by one unit during each step.

Combining these results, the total time for executing the complete and incomplete steps of a job using a greedy schedule is at most $T_1/\overline{P} + T_\infty$. □

The result of Theorem 3 tells us when an execution schedule can achieve ***linear speedup***; that is, when it can achieve execution time $T = \Theta(T_1/\overline{P})$. In particular, we find that this occurs when $\overline{P} \leq T_1/T_\infty$, since then we have $T_1/\overline{P} \geq T_\infty$ and so a greedy schedule executes in time $T \leq 2T_1/\overline{P}$. The quantity T_1/T_∞ is called the ***average parallelism*** of the computation. Thus, if the average effective processor allocation is bounded by the average parallelism, linear speedup occurs.

The proof of Theorem 3 also provides additional intuition for the definition of $\overline{P}$ in Equation (3.1): namely, that we only need to replace the largest T_∞ values of P_t when calculating $\overline{P}$, because there can be at most T_∞ incomplete steps when using a greedy schedule. In more general terms, if a good scheduling algorithm is used to determine the adaptive execution schedule of a job, we expect the job to complete after no more than T_∞ infinite-processor allocations by OS.

Theorem 3 can be easily extended to show the following corollaries.

Corollary 4 *An adaptive greedy schedule yields an execution time that is within a factor of 2 of an optimal schedule.*

Proof. The lower bounds proved in Section 3.1 state that the execution time T of an adaptively parallel job satisfies $T \geq \max(T_1/\overline{P}, T_\infty)$. From Theorem 3, we know

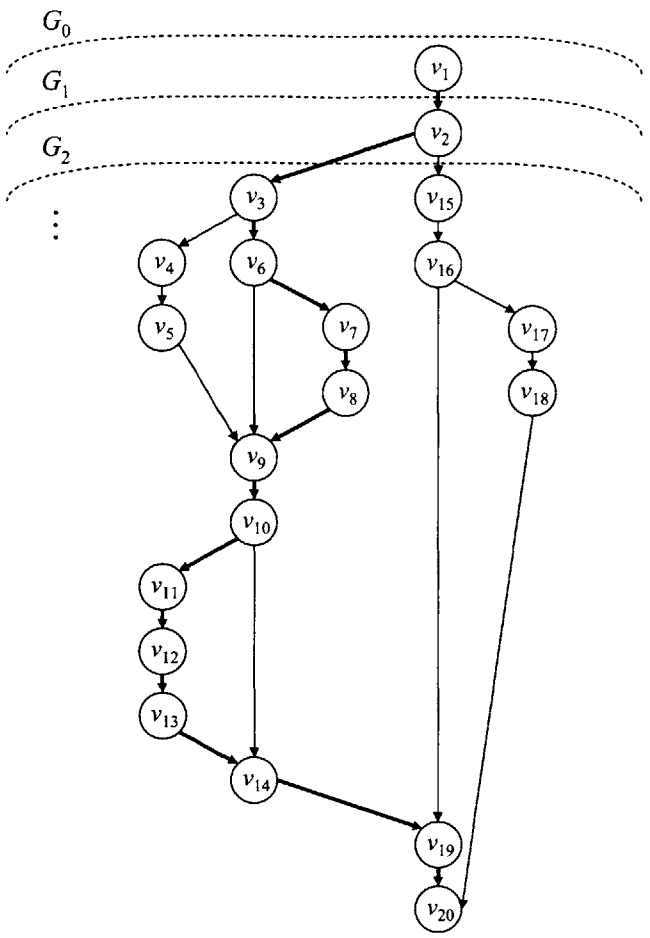


Figure 3-2: The dag corresponding to the computation in Figure 3-1, with the critical path shown in bold. The progression $\langle G_0, G_1, \ldots, G_T \rangle$ shows the subgraph remaining after each incomplete step of a greedy schedule.

that an adaptive greedy schedule yields the following execution time:

$$\begin{aligned} T &\leq \frac{T_1}{\overline{P}} + T_\infty \\ &\leq \max\left(\frac{T_1}{\overline{P}}, T_\infty\right) + \max\left(\frac{T_1}{\overline{P}}, T_\infty\right) \\ &\leq 2\max\left(\frac{T_1}{\overline{P}}, T_\infty\right) \end{aligned} \tag{3.3}$$

Thus, an adaptive greedy schedule is always within a factor of 2 of optimal. □

Corollary 5 *If $\overline{P}$ is much less than the average parallelism T_1/T_∞, then an adaptive greedy schedule achieves almost perfect linear speedup.*

Proof. If $\overline{P} \ll T_1/T_\infty$, then it follows that $T_\infty \ll T_1/\overline{P}$. Thus, the execution time of an adaptive greedy schedule from Theorem 3 reduces to $T \leq T_1/\overline{P} + T_\infty \sim T_1/\overline{P}$. Since the constant in front of $T_1/\overline{P}$ is close to 1, the resulting execution time demonstrates almost perfect linear speedup. □

The argument that maintains Theorem 3 and Corollaries 4 and 5 when multiple adaptively parallel jobs are running in the system is identical to the argument we provide in Section 3.1. In particular, since we treat OS as an adversary and place no constraints on the processor allocation policy it uses, each JS can operate obliviously to the other JS's by simply regarding them as part of OS. In other words, a JS does not need to know how the allocations of its job are determined—including whether or not they are affected by the presence of other jobs—for Theorem 3 to hold.

Chapter 4

A Randomized Work-Stealing Algorithm for Adaptively Parallel Jobs

In Chapter 3, we developed an adversarial model for analyzing the performance of two-level schedulers for adaptively parallel jobs, where the operating system (OS) and job scheduler (JS) play the roles of the first and second-level schedulers, respectively. We showed that if JS uses a greedy algorithm to schedule a job's computation, then the execution time of the job is within a constant factor of optimal. In this chapter, we present another algorithm for JS, called algorithm WSAP, which uses randomized work stealing to schedule the computation of an adaptively parallel job. WSAP is an extension of algorithm WS from [7] that works with dynamic processor-allocation policies. We begin with a description of WSAP in the adversarial model, and then discuss the policy issues surrounding its use in practice. We forego a theoretical analysis of WSAP in favor of investigating its performance practically through the Cilk-AP system (Chapters 6 and 7).

4.1 Algorithm WSAP in the Adversarial Model

In this section, we outline algorithm WSAP, which extends algorithm WS to handle the case where a job's processor allocation changes during its execution. Like WS, WSAP is an online, randomized work-stealing algorithm for scheduling fully strict multithreaded computations; unlike WS, WSAP generates an adaptive execution schedule that conforms to changing processor allocations provided by OS. Since WSAP is an extension of WS, we begin with a reprise of algorithm WS from [7] and [3] and then proceed to extend it.

In algorithm WS (for "work stealing"), the threads of a computation are distributed across a fixed set of processors, each of which attempts to steal work from another processor whenever it runs out of work to do. Each processor maintains a doubly-ended queue, called the ***ready deque***, of threads that are ready to execute; a thread is "ready" if the first unexecuted task in its sequential order is ready. A

processors treats its own ready deque as a stack, pushing and popping threads from the bottom, but treats the ready deque of another processor as a queue, removing threads only from the top. In general, a processor obtains work by removing the bottommost thread of its ready deque, which we call thread A, and executing it until one of three situations occur:

1. *Thread A spawns another thread B.* In this case, the processor pushes A onto the bottom of the deque and starts executing B.
2. *Thread A terminates.* The processor checks the ready deque: if the deque is nonempty, then the processor pops the bottommost thread and begins executing it; if the deque is empty, then the processor tries to execute A's parent thread; if the deque is empty and A's parent is busy, then the processor attempts to work steal, as described below.
3. *Thread A reaches a synchronization point and stalls.* The deque must be empty in this case (if the computation is fully strict), so the processor attempts to work steal.

The work-stealing strategy operates as follows: the processor that is attempting to work-steal, called the ***thief***, chooses another processor uniformly at random, called the ***victim***, and tries to steal work from it. If the victim's deque is nonempty, then the thief removes the topmost thread and starts executing it (the steal is successful); if the deque is empty, the thief restarts the process, choosing another victim at random to steal from (the steal is unsuccessful). Each steal attempt takes one unit of time on a processor. At the beginning of the computation, all the ready deques are empty except for one (the one that contains the root thread), so all but one processor starts out work stealing.

Unlike algorithm WS, algorithm WSAP (for "work stealing, adaptively parallel") schedules a multithreaded computation on a dynamically changing set of processors. In the context of our adversarial model, algorithm WSAP is implemented by JS and responds to changing processor allocations given by OS. Specifically, during each step t of a job's execution, JS receives an allocation P_t from OS that is either greater than, less than, or equal to P_{t-1}, the job's allocation during the previous step. Consequently, JS needs a mechanism to increase or decrease the number of processors being used by the job, or the job's ***usage***, to match the new allocation P_t. In algorithm WSAP, these adjustments are made by sending signals to the relevant processors at the beginning of step t. In particular, a ***sleep signal*** is sent to a processor to reduce the job's processor usage by one, and a ***wake signal*** is sent to a processor to increase the job's usage by one. Thus, at the beginning of each step t, JS takes one of two actions after receiving the allocation P_t:

1. If $P_t < P_{t-1}$, JS sends a sleep signal to any $P_{t-1} - P_t$ processors that are awake.
2. If $P_t > P_{t-1}$, JS sends a wake signal to $P_t - P_{t-1}$ processors that are asleep, waking all processors with nonempty deques first before waking those with empty deques.

For ease of discussion, we assume that a job has an infinite number of "virtual" processors at its disposal, most of which are asleep; during any step t of the job's

execution, algorithm WSAP guarantees that only P_t of the virtual processors are awake. The role of OS thus reduces to providing a number P_t to JS, instead of actually allocating the physical processors. This abstraction does not violate the rules of our adversarial model, since we guarantee that JS only uses as many processors as are allocated, but makes it easier for us to deal with processors that are put to sleep in the middle of working, since these processors cannot be returned to the system without some protocol for handling their (nonempty) deques. We overload the terms "virtual processor" and "processor" in the remainder of this chapter.

The processors in algorithm WSAP follow the same rules of operation as the processors in algorithm WS, except now they need to respond appropriately to sleep and wake signals. In particular, we add a fourth situation to the three situations listed above that can interrupt a processor's execution:

4. *The processor receives a sleep signal from JS at the beginning of step t.* In this case, the processor immediately puts itself to sleep, regardless of whether it is executing a task or attempting to steal work during step t. The processor remains asleep until it receives a wake signal from JS or another processor at the beginning of step $t' > t$, at which point it continues executing where it left off.

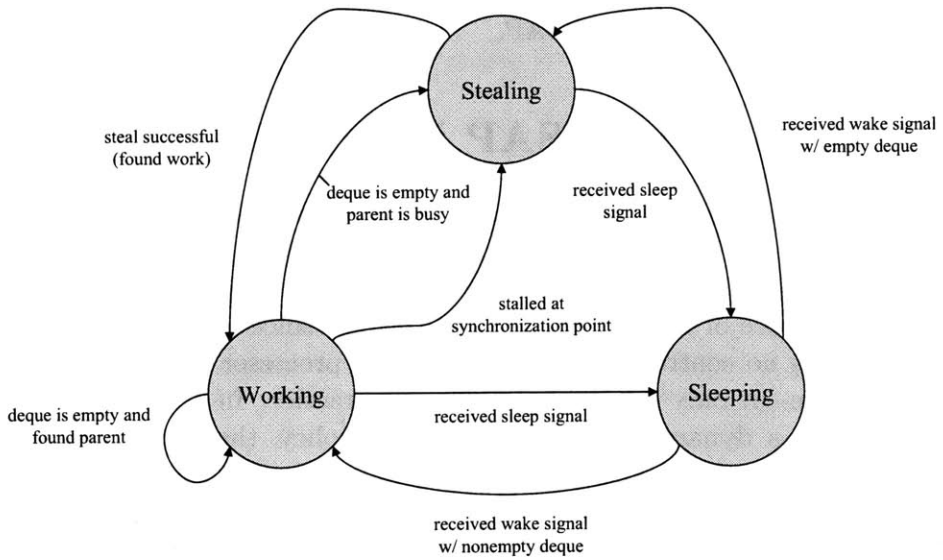


Figure 4-1: State diagram of a processor's execution in algorithm WSAP.

Figure 4-1 summarizes the execution of a processor in algorithm WSAP using a state diagram; the three possible states a processor can be in are "Working" (when the processor is executing a thread), "Stealing" (when the processor is making a steal attempt), and "Sleeping" (when the processor responds to a sleep signal). Since processors can receive sleep signals at the beginning of any step, it is possible for some of them to go to sleep even if they have work on their deques. To prevent this

work from going unnoticed, we modify the work-stealing strategy as follows: the thief processor chooses a victim from the set of processors that are either awake or sleeping with nonempty deques. If the victim processor is asleep, then the thief sends a wake signal to the victim and puts itself to sleep (note that the thief has an empty deque since it is work stealing); the victim awakens and continues executing where it left off. If the victim processor is awake, then the thief follows the original work-stealing protocol: if the victim's deque is nonempty, the thief removes the topmost thread and starts executing it, and if the deque is empty, the thief restarts the process and chooses another victim at random.

Using algorithm WS, the expected time to execute a multithreaded computation on P processors is $O(T_1/P + T_\infty)$, where T_1 is the work and T_∞ is the critical-path length. This bound has been shown using a delay-sequence argument by Blumofe and Leiserson [7], as well as using a potential-function argument by Arora et. al. [2]. Arora et. al. also prove an execution time bound for a nonblocking implementation of WS in a two-level scheduling environment, where the computation executes on a fixed set of P processes that are scheduled by the operating system onto a time-varying allocation of physical processors. In the case of algorithm WSAP, the mapping between processes to physical processors is one-to-one; that is, the computation executes on exactly as many virtual processors as there are physical processors. We expect that one can use an approach similar to the potential-function argument used in [2] to analyze the execution time of algorithm WSAP.

4.2 Algorithm WSAP in Practice

In the adversarial model, the decisions made by OS are completely decoupled from those made by JS; specifically, OS does not consult with JS when determining a job's allocation for a given time step. While this model is suitable for analyzing the worst-case performance of a job scheduling algorithm—since it captures the notion that a given job has no control over the number and processor requirements of other jobs in the system—it only tells half the story in practice. In a real two-level scheduling system using a dynamic processor-allocation policy, the operating system and job scheduler communicate with each other to determine the current allocation of a job. In particular, the operating system gathers information about the current processor desire of each job in the system, and then uses this information to make its allocation decisions. While there exist dynamic processor-allocation policies that do not use a job's current processor desire in their decisions, these policies do not respond to changes in the parallelism of jobs during runtime, and thus cannot run adaptively parallel jobs efficiently (see Chapter 5).

In order to estimate the processor desire of a job during runtime, there are three policy questions that need to be addressed:

1) who is responsible for estimating a job's processor desire,
2) how should the desire be estimated, and
3) how often should the desire be estimated.

We discuss the policies used in algorithm WSAP in response to these questions in the remainder of this section. We begin by illustrating a fundamental problem with strategies that measure a job's instantaneous parallelism directly, and use this problem to guide our policy decisions on questions 1) and 2). Then, we explain our policy of estimating a job's desire at regular intervals instead of irregular ones, in response to question 3).

Estimating a Job's Processor Desire

In order to provide a good estimate for the processor desire of a job, we need information about the different runtime characteristics of the job, such as the number of threads it is currently using, the amount of work being performed by those threads, or any synchronization constraints that limit processor utilization. Since this information is directly and readily available to the job scheduler, we give the responsibility of estimating a job's processor desire to the job scheduler itself (i.e. the second-level scheduler). While it is true that the operating system can obtain different runtime statistics of a job—such as the utilization of the job's processors or the size of its memory footprint—this information is necessarily coarse-grained and reveals nothing about the internal workings or dependencies of the job. For example, consider a program written in the Cilk multithreaded language [47], and recall that Cilk uses a work-stealing scheduler which implements algorithm WS. During a given run of the program, the operating system can only determine the number of processors being used by the job and has no direct knowledge about the number of threads that have been queued on the ready deques of the processors. Since each of these threads could potentially execute in parallel, it is important that the entity responsible for estimating the job's desire be aware of them. The only entity with direct access to this information is the job scheduler.

Given that the job scheduler is in charge of estimating a job's processor desire, the question still remains as to how this estimate is made and, in particular, what runtime information about the job is used. As we observed in Chapter 2, most prior work on dynamic processor-allocation policies assumes that the processor desire of a job is either known by the scheduling system or inferred using some measurement of the job's instantaneous parallelism. In the latter case, almost all studies use the number of operating system threads or processes being used by the job to measure its instantaneous parallelism. Even if this method were accurate, however—which, as we showed in the example of Cilk above, is not always the case—we maintain that *any measure of a job's instantaneous parallelism is not an accurate or reliable measure of its processor desire.* To see why this is true, we present a typical example of a multithreaded computation whose instantaneous parallelism is always bounded by the number of processors it is allocated, but whose average parallelism may be orders of magnitude higher. This computation, called the LOOPY benchmark, is shown in Figure 4-2. We assume that LOOPY is scheduled using a depth-first algorithm (like algorithm WS), meaning that execution goes to the child whenever a thread is spawned or created.

Suppose that LOOPY starts out executing on one processor. The processor starts

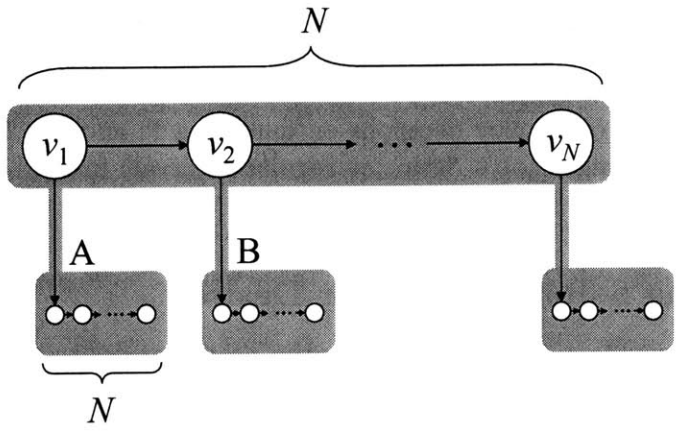


Figure 4-2: The LOOPY benchmark: a multithreaded computation with work $T_1 = N^2 + N$ and critical-path length $T_\infty = 2N$.

at task v_1 of the root thread but then immediately spawns thread A, thus beginning working on A. Assuming that N is very large, each of the subcomputations in threads A, B, etc. may take a long time to complete, and so we assume that the processor is "stuck" on thread A for all practical purposes. At this point, the instantaneous parallelism of the job is 2, as measured by the number of threads it has created (the root thread and A). If another processor is allocated to the job, this processor can continue executing the root thread while the first processor executes A, but it gets stuck working on thread B almost immediately. The instantaneous parallelism of the job is now 3 (the root thread, A, and B). The process continues as more processors are allocated to the job. At any given time, the instantaneous parallelism of the job is only 1 greater than its allocation. Since all N subcomputations spawned by the root thread of LOOPY can theoretically execute in parallel, the instantaneous parallelism is a gross underestimate of the average parallelism of the job. Specifically, the average parallelism of the LOOPY is

$$\begin{aligned} \frac{T_1}{T_\infty} &= \frac{N^2 + N}{2N} \\ &= \frac{N+1}{2} . \end{aligned}$$

If N is large, the average parallelism could be orders of magnitude greater than the instantaneous parallelism. Moreover, an execution of LOOPY achieves linear speedup as long as $P \leq (N+1)/2$. For $N > 10^5$, this number is larger than most (if not all) multiprocessor machines in existence today.

The LOOPY benchmark represents a common class of data-parallel programs, in which a single loop in the main thread spawns a large number of equal-sized subcomputations. The following code fragment, for example, produces a LOOPY computation:

```
for (i = 0; i < N; i++) {
  spawn Work_N(i);
}
```

where `Work_N` is a subcomputation with N units of work. Programs of this form provide a simple and effective means of generating large amounts of parallelism in a computation. They are especially common in parallelized versions of serial code. Their widespread use reinforces our case against policies that use instantaneous parallelism measures to estimate a job's processor desire.

In light of this example, we seek a policy for estimating a job's desire that more accurately gauges its average parallelism, but also responds to periods of high (or low) levels of parallelism that may occur at arbitrary times during the job's execution. Specifically, since we cannot reliably discern the potential parallelism of a job through instantaneous measurements, we employ a policy that proactively explores its "future" parallelism and uses runtime information about the job to tune the estimated desire. The basic idea is to give a job more processors than it apparently needs at the current time, and then remove processors that aren't being used efficiently. Using this strategy, we can find and exploit sources of parallelism that may be hidden in the current state of the computational dag, but which reveal themselves as the dag is unfolded. (Recall that since we are studying online scheduling algorithms, we only discover the structure of the dag as it unfolds.)

In algorithm WSAP, the primary mechanism for exploring the parallelism of a job is by work stealing: when a processor steals work from its victim, it removes and executes the topmost thread from the victim's deque, which is necessarily an ancestor of the thread currently being executed by the victim (see Lemma 4 in [7]). Since the ancestor thread is shallower (in terms of dag depth) than the thread being executed by the victim, the thief is able to expose parallelism starting at a higher level in the computational dag. In the LOOPY benchmark, for example, the second processor allocated to the job steals the root thread from the deque of the first processor, thus continuing the execution of the main-level loop and exposing more of the job's parallelism through thread B. Evidently, it makes sense to use runtime information about steal attempts to estimate a job's processor desire. In particular, if the number of steal attempts is too low, then we increase the estimated desire of the job, even if the amount of work available on the ready deques doesn't seem to warrant such an increase; if the number of steal attempts is too high, then we reduce the job's desire. The underlying intuition is that, if there are many unsuccessful steal attempts occurring, then there are too many processors being used inefficiently by the job, whereas if the number of unsuccessful steal attempts is low, then most processors are being used efficiently, so we can afford to increase the allocation in search of additional parallelism. We formalize the notions of efficiency and inefficiency—as they relate to a job's usage of processors—in Section 6.1. The intuition presented here is justified theoretically in previous (unpublished) work by the Supercomputing Technologies Group at MIT. We discuss this work in Chapter 8.

When using the number of unsuccessful steal attempts to measure an allocation's inefficiency, we focus on those attempts that are ***purely unsuccessful***, meaning that

the victim processor is in the middle of work stealing itself, as opposed to attempts that are ***partly unsuccessful***, where the victim processor has an empty deque but is busy working on some thread. Purely unsuccessful steal attempts give a better indication of the number of inefficiently used processors in a job, since both thief and victim processors are looking for work. In a partly unsuccessful steal attempt, however, the victim processor is actually doing work, so it is misleading to use such attempts to directly measure the inefficiencies of a job's allocation. For example, if the LOOPY benchmark is scheduled using algorithm WSAP, then the total number of unsuccessful steal attempts is always very large, because only one thread can be stolen at any given time; the fraction of these attempts that are purely unsuccessful, however, is very low, because most processors are busy working on a subcomputation when their deques are empty. As a result, any decisions based on the total number of unsuccessful steal attempts—or the number of partly unsuccessful steal attempts—would lean towards reducing the job's allocation, even if all the processors are busy doing work (and hence being used efficiently). Decisions based on the number of purely unsuccessful steal attempts, on the other hand, would lean towards increasing the job's allocation, thus exploiting the high average parallelism of the computation.

The details of what thresholds to use when comparing the number of unsuccessful steals attempts, and how much to increase or decrease the allocation by, are left to the particular implementation of algorithm WSAP. We discuss these decisions for the Cilk-AP system in Section 6.1.

Frequency of Desire Estimations

We have established that the job scheduler is responsible for estimating the processor desire of a job and that it does so using runtime information about unsuccessful steal attempts. The question remains, however, as to how often these estimations should be made. We consider two options. Option 1 is to estimate the job's desire at evenly spaced intervals, where the size of the interval can be tuned either statically or dynamically. Option 2 is to estimate the job's desire at irregular intervals, perhaps corresponding to relevant events in the job's execution (for example, when a processor successfully steals work, or conversely, when a processor is repeatedly unable to steal work).

There are several disadvantages to Option 2. First, performing desire estimations at important events in the job's execution may add an unacceptable amount of overhead. In the case of Cilk jobs, for instance, the work-first principle [19] states that scheduling overheads borne by the work of a computation should be minimized, since they contribute more to the execution time of a job than overheads borne by the critical path. Thus, if processors that are currently working are expected to perform desire-estimation calculations, the effect on the job's running time may be adversely large. Even if this overhead is reduced—for example, by restricting the estimation calculations to processors that are in the middle of stealing (adding to critical-path overhead instead of work overhead)—the irregularity of the estimation intervals makes it difficult to measure or tune the incurred overhead, let alone reason about it.

As a result, we stick to Option 1 in algorithm WSAP, performing the desire es-

timation at regular intervals. By tuning the size of the interval, we can balance the trade-off between the overhead incurred by estimating a job's desire and the responsiveness of these estimates to changes in the job's parallelism. Specifically, larger intervals reduce the overhead incurred by the desire estimation, while reducing the responsiveness of the estimates to changes in the job's parallelism. Smaller intervals achieve the opposite effect, incurring greater overhead but responding to changes in parallelism more promptly. We discuss our approach to tuning this interval in the context of the Cilk-AP system in Section 6.1.

Chapter 5

A Dynamic Processor-Allocation Algorithm for Adaptively Parallel Jobs

In a real two-level scheduling system using a dynamic processor-allocation policy, the operating system communicates with the job scheduler to determine the current allocation of a job. In Section 4.2, we discussed the policies used by algorithm WSAP in the second level of scheduling to estimate the processor desire of a job and report it to the first level. In this chapter, we describe a well-known dynamic processor allocation algorithm, algorithm DP, which allocates processors to multiple, concurrent, adaptively parallel jobs in the first level of scheduling. DP uses information about the processor desires of each job to distribute processors in a way that is fair, efficient, and conservative. We begin by defining these three conditions, and then we show how DP satisfies them. Algorithm DP is used in the first-level scheduler of the Cilk-AP system in Chapter 6.

5.1 Properties of a Good Dynamic Processor-Allocation Algorithm

In this section, we define three conditions for a good dynamic processor allocation algorithm: fairness, efficiency, and conservatism. Together, these conditions ensure that an allocation achieves both low job response times and high job throughput.

We consider a parallel system with P processors and J running jobs, where $J \leq P$. At any given time, each job $j = 1, 2, \ldots J$ has a processor desire d_j, representing the maximum number of efficiently usable processors, and an allotment a_j, representing the number of processors allocated to it. If $a_j < d_j$, we say that the job is ***deprived***, since it has fewer processors than it desires; if $a_j = d_j$, we say that the job is ***satisfied***, since its desire has been met. We define the notions of fairness, efficiency, and conservatism as follows:

1. An allocation is ***fair*** if the processors are distributed equally among the jobs.

Mathematically, if there exists a job j such that $a_j < d_j$, then for all $i = 1, 2, \ldots, J$, $m_i \leq a_j + 1$; whenever a job is deprived, then no other job receives more than 1 more processor than this job receives. (The allowance of one processor is due to integer roundoff.)

2. An allocation is ***efficient*** if it uses as many processors as possible. Mathematically, if there exists a job j such that $a_j < d_j$, then $\sum_{j=0}^{J} a_j = P$; if a job is deprived, then there must be no free processors in the system.
3. An allocation is ***conservative*** if no job receives more processors than it desires. Mathematically, for all $j = 1, 2, \ldots J$, $a_j \leq d_j$; a job is either deprived or satisfied.

The fairness condition allows all jobs to run simultaneously, instead of having some of them wait in a queue. The conservatism condition ensures that jobs use their allocations efficiently (by not giving processors to jobs that do not use them well), and the efficiency condition ensures that the system utilizes all processors efficiently. As long as the efficiency condition is met, both fairness and conservatism contribute to high job throughput. Furthermore, since jobs do not spend arbitrarily long periods of time in queues, the response times of jobs more directly reflect the amount of computation they have to perform.

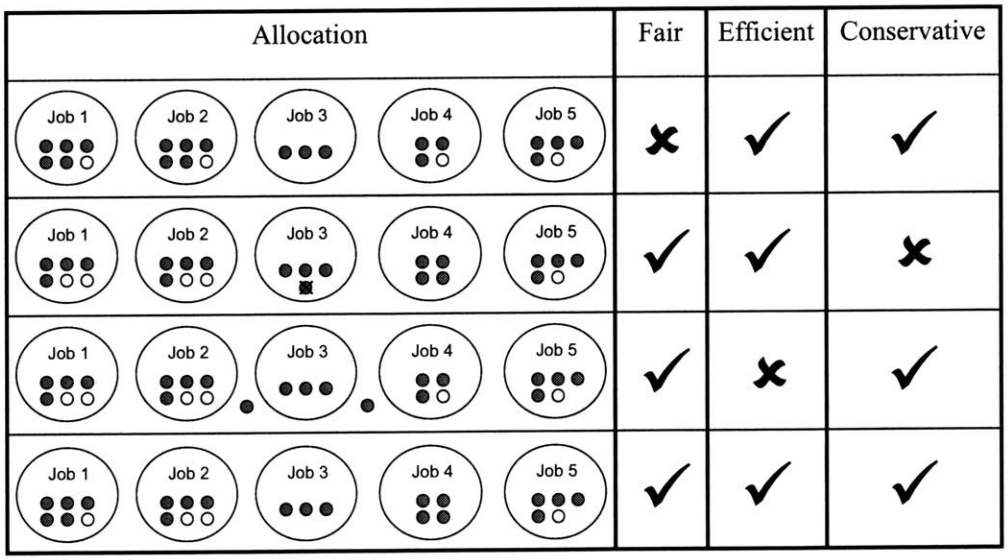


Figure 5-1: Example allocations for a system with $P = 20$ processors and $J = 5$ jobs; the fairness, efficiency, and conservatism conditions are evaluated on the right. Each large circle represents a job, and the small circles within the job represent its processor desire; small circles that are grayed represent actual processors. The gray circle that is crossed out in Job 3 of the second allocation represents a processor allocated to the job beyond its desire.

Figure 5-1 shows examples of different allocations for a system with $P = 20$ and $J = 5$. For each allocation, we determine whether or not it satisfies the conditions

of fairness, efficiency, and conservatism. In particular, the first allocation is not fair, because Job 4 is deprived and receives only 2 processors while other jobs receive more than 3. The second allocation is not conservative, because Job 3 is given one more processor than it desires. The third allocation is not efficient, because there are free processors in the system that are not being used to satisfy deprived jobs. The fourth allocation satisfies all three conditions.

5.2 Algorithm DP

In this section, we present algorithm DP (for "dynamic partitioning"), a well-known dynamic processor-allocation algorithm that is fair, efficient, and conservative. DP employs a dynamic version of the basic ***equipartition*** policy [33, 48], and hence is sometimes called "dynamic equipartitioning".

The equipartition policy strives to maintain an equal allotment of processors to all jobs, with the constraint that a job's request is an upper bound on the number of processors it receives. During a reallocation, processors are distributed among the jobs as follows: each job starts out with 0 processors, and the allotment of each job is incremented by 1 in turn, where jobs that are satisfied drop out of the allocation. The process continues until either all jobs have dropped out or P processors have been allocated. We recognize three types of equipartitioning. In static equipartitioning, reallocations are not allowed to change the allotments of existing jobs, and so imbalances in the allocation and queuing may occur. In regular equipartitioning, reallocations occur on job arrival and completion only, and so the allocation is fair, efficient, and conservative at all times if we assume that the processor desires of jobs remain fixed during their execution. In dynamic equipartitioning, reallocations can occur at any time—responding to both job arrivals and completions as well as changes in the processor desires of jobs during execution—and so the allocation is always fair, efficient, and conservative.

Although dynamic partitioning can be implemented using the equipartitioning algorithm described above, we present a more practical version for algorithm DP that does not have to redistribute processors every time a reallocation occurs. Assume that there are no jobs in the system initially, and that jobs arrive one at a time (i.e. no two jobs arrive at exactly the same time). We define the ***fair share*** of a job to be the quantity:

$$\texttt{fair\_share} = \frac{P - \sum_{S=\{j \mid d_j < \lfloor P/J \rfloor\}} a_j}{J - |S|} \tag{5.1}$$

where P is the total number of processors and J is the number of jobs in the system. In other words, a job's fair share is computed by discounting all processors being used by jobs that desire less than the system equipartition $\lfloor P/J \rfloor$ and then distributing the remaining processors equally among jobs that desire more than $\lfloor P/J \rfloor$. Thus, when a new job j with initial desire d_j and initial allocation $a_j = 0$ arrives into the system, algorithm DP operates as follows:

Current Allocation State	Input Event	Ending Allocation State
$\{\}$	Arrival: job 1 with $d_1 = 4$	$\{4\}$
$\{4\}$	Arrival: job 2 with $d_2 = 16$	$\{4, 12\}$
$\{4, 12\}$	Arrival: job 3 with $d_3 = 2$	$\{4, 10, 2\}$
$\{4, 10, 2\}$	Change in desire: $d_3 = 16$	$\{4, 6, 6\}$
$\{4, 6, 6\}$	Arrival: job 4 with $d_4 = 8$	$\{4, 4, 4, 4\}$
$\{4, 4, 4, 4\}$	Arrival: job 5 with $d_5 = 8$	$\{3, 3, 3, 3, 4\}$
$\{3, 3, 3, 3, 4\}$	Arrival: job 6 with $d_6 = 8$	$\{2, 2, 3, 3, 3, 3\}$
$\{2, 2, 3, 3, 3, 3\}$	Completion: job 2	$\{3, 4, 3, 3, 3\}$
$\{3, 4, 3, 3, 3\}$	Completion: job 3	$\{4, 4, 4, 4\}$
$\{4, 4, 4, 4\}$	Completion: job 6	$\{4, 6, 6\}$

Figure 5-2: A sample allocation trace for algorithm DP on a 16-processor system.

1. Let `free_procs` be the number of free processors in the system at the time when j arrives. If `free_procs` $\geq d_j$, set $a_j = d_j$. Otherwise, set $a_j =$ `free_procs` and go to step 2.
2. Compute the value of `fair_share` given by Equation (5.1) (include job j in the calculation). If $a_j \leq \min(d_j,$ `fair_share`$)$, remove one processor from a job that has the highest allocation and give this processor to j.
3. Repeat step 2 until either $a_j = d_j$, meaning that j is satisfied, or $a_j =$ `fair_share`, meaning that j now has its fair share of processors.

The same steps are performed by DP when the processor desire of an existing job increases; the only difference is that a_j does not start out at 0 in this case. When the desire of a job j decreases, or when j completes, DP operates as follows:

1. Add the number of freed processors to `free_procs`. Recompute `fair_share` using Equation (5.1) if j has completed.
2. Add a processor to a deprived job that has the lowest allocation.
3. Repeat step 2 until either `free_procs` $= 0$ or there are no deprived jobs.

Figure 5.2 shows a sample allocation trace for algorithm DP on a system with 16 processors. Each row of the table shows the current state of the allocation (expressed as a list of allotments), an input event (either a job arrival, job completion, or change in a job's processor desire), and the ending state of the allocation.

Several studies have shown that algorithm DP outperforms all other space-slicing policies when the reallocation overhead is low, regardless of job workload or overall system load [22, 27, 33, 39, 46, 48, 52]. Most of these studies focus on uniform-access, shared-memory (UMA) machines, where the allocation of processors is decoupled from the allocation of memory (see Chapter 2). Some studies have also shown that DP performs better than static allocation policies under realistic reallocation overheads [15], and for some classes of workloads, under a wide range of overheads [52]. In general, the advantage of using algorithm DP increases with larger and more rapid changes in the parallelism of the workload, as well as increasing system load [52].

The performance of DP decreases, however, when the number of jobs in the system exceeds the number of processors (i.e. during high loads), since some of the jobs must be queued [35, 52]. While queuing is not expected to be a problem in large parallel machines [17], various strategies can be used to reduce its effect on the response time of jobs [16,42,43,49]. A mathematical model for algorithm DP, including an analysis of its performance across a wide range or parallel system environments, is provided in [46]. Examples of real implementations of DP can be found in [12,32].

In practice, there are a few policy decisions that need to be made when implementing algorithm DP, most of which relate to the way reallocations occur. The first decision is between uncoordinated reallocations, where the operating system reallocates processors without interacting with the job scheduler, and coordinated reallocations, where processors are reallocated in concert with the job scheduler. We choose the policy of coordinated reallocations for reasons discussed in Chapter 2. Given that reallocations are coordinated, it is still unclear how much responsibility or control the job scheduler should have during a reallocation, as well as how much trust this responsibility requires between the operating system and job scheduler. We also need to determine when and how often reallocations occur—which may or may not depend on how often jobs report their processor desires (see Section 4.2)—so that we can control the amount of overhead a reallocation incurs. In practice, reallocations never occur instantaneously (this is true of both uncoordinated and coordinated policies), and so there is necessarily a delay between when a job's allotment is changed by the operating system and when its processor usage matches the new allotment. If p_j represents the number of processors currently being used by job j, then there is a period of time when $p_j > a_j$ after j's allotment is decreased, and a period of time when $p_j < a_j$ after j's allotment is increased. Also, we need to decide how to deal with situations where $J > P$ (e.g. whether to multiplex the existing processors or just queue the excess jobs), keeping in mind that queuing can dramatically affect the performance of algorithm DP relative to other processor-allocation policies [52]. We discuss all of these policy decisions and issues as they pertain to the Cilk-AP system in Chapter 6.

Chapter 6

Cilk-AP: A Two-Level Scheduler for Adaptively Parallel Work-Stealing Jobs

In this chapter, we present the design and implementation of Cilk-AP, a two-level scheduling system for adaptively parallel work-stealing jobs running on a multiprogrammed shared-memory system. In the first level of scheduling, Cilk-AP uses algorithm DP from Chapter 5 to allocate processors to jobs in a fair, efficient, and conservative manner. In the second level of scheduling, Cilk-AP uses algorithm WSAP from Chapter 4 to schedule the computation of a single job and report its processor desire to the first-level scheduler. The Cilk-AP system is implemented by extending the runtime system of the Cilk multithreaded language [5,19,47]. We first present an overview of the design of Cilk-AP, addressing some of the policy questions raised in Chapters 4 and 5. Then, we describe its implementation.

6.1 Design Overview

Figure 6-1 provides an overview of the Cilk-AP system. Cilk-AP uses a global allocation table (GAT) to maintain a fair, efficient, and conservative allocation of processors to jobs in the first level of scheduling. The allotments in the GAT are computed using algorithm DP, and the GAT itself is stored in shared memory. In the second level of scheduling, Cilk-AP uses algorithm WSAP to schedule the computation of a job and estimate its processor desire. The estimates are performed at regular intervals and entered directly into the GAT. In Cilk-AP, both the first and second levels of scheduling are performed by the job schedulers of the running jobs. In this section, we describe the three major functions performed by the job scheduler:

1) estimating the processor desire of a job,

2) maintaining a fair, efficient, and conservative allocation, and

3) adjusting the processor usage of a job to match its allotment.

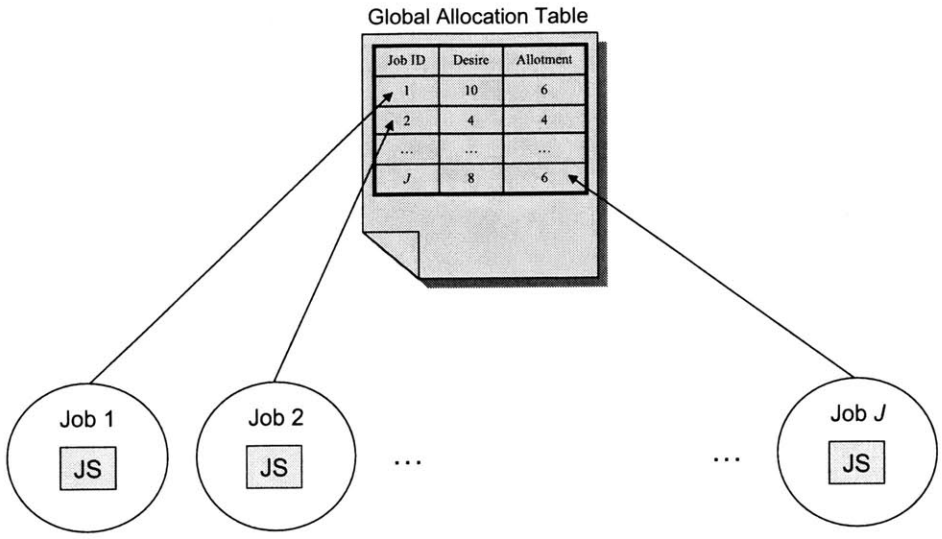


Figure 6-1: Overview of the Cilk-AP system. Jobs record their processor desires in the GAT and recompute the allocation for the entire system if necessary.

We end by discussing our strategy for tuning the parameters used in the above functions. In the course of our discussion, we present the policy decisions of the Cilk-AP system in response to the questions and issues raised in Sections 4.2 and 5.2 (regarding the use of algorithms WSAP and DP in practice, respectively).

Estimating a Job's Processor Desire

The circles in Figure 6-1 represent multiple, concurrent, adaptively parallel jobs running in the Cilk-AP system. Each job has an associated job scheduler that schedules the job's computation using algorithm WSAP. In accordance with the policies of WSAP (Section 4.2), the job scheduler is also responsible for estimating the processor desire of the job and reporting this desire to the GAT; in other words, job desires are estimated in the second level of scheduling. Let d_j be the current processor desire of job j and let p_j be the number of processors currently being used by j (the distinction between p_j and a_j, the job's allotment, is explained in Section 5.2). We define the ***efficiency*** of j to be the ratio of processors that are busy working to p_j; the ***inefficiency*** of j, therefore, is the ratio of processors that are busy stealing to p_j. The desire-estimation process occurs as follows: at any given time in j's execution, each of the p_j processors keeps track of the number of purely unsuccessful steal attempts and the number of total steal attempts that take place on the processor (recall from Section 4.2 that purely unsuccessful steal attempts are a better measure of an allocation's inefficiency than partly unsuccessful steal attempts). Every `est_cycle` seconds, the job scheduler uses the steal-attempt statistics from each processor to approximate the overall inefficiency of j, and then uses this approximated value to estimate d_j. More specifically, the job scheduler performs the following operations

every `est_cycle` seconds:

1. Calculate the ratio of purely unsuccessful steal attempts to total steal attempts across all p_j processors, using

$$\texttt{PUS\_ratio} = \frac{\sum_{i=0}^{p_j} \text{no. of purely unsuccessful steal attempts on processor } i}{\sum_{i=0}^{p_j} \text{no. of total steal attempts on processor } i}.$$

   Reset the steal-attempt counters on all p_j processors to start the next interval of counting.
2. If $\texttt{PUS\_ratio} \leq 1-\eta$, where $0 < \eta \leq 1$ is the target efficiency, set $d_j = (1/\eta) \cdot p_j$; otherwise, set $d_j = \lceil ((1 - \texttt{PUS\_ratio})/\eta) \cdot p_j \rceil$.
3. Replace the old value of d_j stored in the GAT with the new value computed in step 2.

Since the GAT is stored in shared memory, the job scheduler is able to directly update the row corresponding to j in step 3; other job schedulers perform similar updates, as illustrated by the arrows in Figure 6-1. The interval at which the above operations occur, `est_cycle`, is a tunable parameter that balances the overhead of the desire-estimation process with its responsiveness to changes in a job's parallelism. The value of `est_cycle` is set by the job scheduler and may vary from job to job. The parameter η in step 2, $0 < \eta \leq 1$, is another tunable parameter that represents the target efficiency of job j. The value of η can be set by either the user or the job scheduler and can also vary from job to job. We discuss the tuning of the parameters `est_cycle` and η at the end of Section 6.1.

Our definition of efficiency for adaptively parallel work-stealing jobs is based on the premise that working processors are efficient because they make progress on the total work (T_1) of a job's computation. Stealing processors, on the other hand, do not contribute to T_1, and are therefore inefficient. By setting a target efficiency η for job j, we effectively set a limit on the fraction of processors that can be stealing at any given time, or the inefficiency of j. Since $1-\eta$ represents the target value for this fraction, it also represents the probability that a victim processor chosen uniformly at random during a steal attempt is also busy stealing. Thus, we can use the ratio of purely unsuccessful steal attempts (those in which the victim processor is busy stealing) to total steal attempts in a given interval to approximate the inefficiency of j during that interval. (This value is calculated in step 1 as `PUS_ratio`.) If `PUS_ratio` is less than the target inefficiency $(1-\eta)$, then we treat the ratio as effectively 0 and set j's desire to $(1/\eta$ times its current processor usage, or the usage at which j's efficiency drops to η (and its inefficiency rises to $1-\eta$) if j is 100% efficient to begin with. If the inefficiency is greater than $1-\eta$, then we set j's desire to the fraction of the current usage that matches the target inefficiency, which we derive in the next paragraph.

The reason we overestimate j's desire when its current inefficiency drops below $1-\eta$ is to allow us to proactively explore the future parallelism of j, in accordance with our policies from Section 4.2. By approximating j's inefficiency to 0, we temporarily

overstate its processor desire in the hopes of finding more work and exploiting any available parallelism sooner. While this boost in processors temporarily reduces j's efficiency below η, the hope is that a sufficient number of the added processors are able to find work before the next interval, thus increasing the overall efficiency of j. Also, there are two additional safeguards that prevent j's inefficiency from greatly exceeding $1 - \eta$ during this period. The first safeguard is that we only increase j's desire by a factor of $(1/\eta)$ over its current usage, so the resulting efficiency is lower bounded (at least initially) by $(1/\eta^2)$; if $\eta > 0.5$, then the efficiency drops by at most a factor of $1/2$. The second safeguard is that we only increase j's processor desire d_j, and not its actual allotment a_j. The allotment of j is determined by the first-level scheduler of Cilk-AP, and is guaranteed never to exceed the fair share of j. Thus, there is no risk that a large desire reported by the job scheduler can monopolize the available processors in the system or detract from the fair shares of other jobs.

If, despite these two safeguards, the processors of j are unable to find sufficient work to increase j's efficiency to η, then the desire of j is reduced in step 2 during the next `est_cycle` interval. In particular, if j's inefficiency is greater than $1 - \eta$, then we reduce its desire to avoid wasting processors that may be used more efficiently by other jobs. Unlike increases to j's desire, however, we do not want to overstate this reduction, since an underestimation of the desire can adversely affect the response time of j. Thus, a good compromise is to try to match the target inefficiency exactly, by calculating the number k of stealing processors that need to be removed in order to lower j's inefficiency to $1-\eta$. Since `PUS_ratio` is the approximation of j's inefficiency during the most recent interval, and p_j is the usage of j during that interval, we can calculate k as follows:

$$\frac{\texttt{PUS\_ratio} \times p_j - k}{p_j - k} = 1 - \eta \tag{6.1}$$

Multiplying both sides of Equation (6.1) by $(p_j - k)$ and solving for k, we have:

$$\begin{aligned} k &= \frac{p_j(1-\eta) - \texttt{PUS\_ratio} \times p_j}{1 - \eta - 1} \\ &= \left(\frac{\texttt{PUS\_ratio} + \eta - 1}{\eta}\right) p_j \ . \end{aligned}$$

Subtracting this value of k from p_j yields the new desire of j shown in step 2. Figure 6.1 shows a sample desire-estimation trace for job j with $\eta = 0.5$ running on a 16-processor system, where one other job runs concurrently with j for the duration of the trace. Each row of the table shows the values of `PUS_ratio`, p_j, d_j, and a_j after the desire-estimation process has occurred during the corresponding `est_cycle` interval. The value of a_j is set by the first-level scheduler and never exceeds the fair share of j, as computed by Equation (5.1). (We assume that the second job in the system always desires more than the system equipartition, or $\lfloor P/J \rfloor = \lfloor 16/2 \rfloor = 8$ processors.)

One aspect of the desire-estimation process that merits further discussion is the manner in which we measure a job's efficiency in step 1 by approximating its inef-

Interval no.	PUS_ratio	p_j	d_j	a_j
1	0.15	4	8	8
2	0.45	8	16	8
3	0.70	8	5	5
4	0.55	5	5	5
5	0.50	5	10	8
6	0.95	8	1	1
...	...	...	...	...

Figure 6-2: A sample desire-estimation trace for job j with $\eta = 0.5$. There are 16 processors in the system and one other job running concurrently with j whose desire is consistently greater than 8 processors.

ficiency using the ratio of purely unsuccessful steal attempts across all processors. There are two main reasons why we measure the inefficiency of the job instead of directly measuring its efficiency (for example, by counting the number of partly unsuccessful steal attempts or successful steal attempts): relevance and overhead. Since steal attempts are made by stealing processors, they are only accurate indicators of periods of inefficiency in the job's execution. In other words, when most of the job's processors are stealing—that is, when the job is being inefficient—there are a large number of steal attempts, and so any measurements based on steal-attempt statistics are accurate. When most of the job's processors are working, there are few steal attempts, so any measurements we take are inherently inaccurate. Since we are trying to approximate the job's inefficiency, however, it is not very important if our measurements are inaccurate when most of the job's processors are working, because then the job is being efficient and its inefficiency is low. Thus, by choosing to measure the inefficiency of the job, the accuracy of our measurements match their degree of relevance to the current situation. If we choose to measure the efficiency of the job directly, the circumstances would be reversed, and we would not achieve the desired level of accuracy in our measurements. Furthermore, when measuring the inefficiency of the job, the overhead of maintaining the steal-attempt statistics also matches their degree of importance: when the job is being inefficient, we spend more time collecting steal-attempt information because of the large number of steal attempts. When the job is being efficient, however, we incur little overhead, because the number of steal attempts is very low. Again, this situation would be reversed if we were trying to measure the job's efficiency directly.

Another alternative to our method of approximating a job's inefficiency is to have the job scheduler poll all p_j processors at regular intervals and check which ones are working or stealing. We decided against this strategy for two reasons: compatibility with the existing scheduling algorithm and overhead. Since all jobs are scheduled using algorithm WSAP, our strategy of maintaining a few steal-attempt counters while work stealing is a simple and unobtrusive extension to the algorithm. The current implementation of the Cilk system, for example, already has this capability built into its work-stealing scheduler. Furthermore, since we only take measurements

while work stealing, the overhead incurred by our strategy is always borne by the critical path of the computation (satisfying the work-first principle [19]), and is always proportional to the inefficiency of the job. In a regular polling strategy, the amount of overhead remains fixed throughout the job's execution, and it is not clear where the overhead is borne. If the job spends most of its time using its processors efficiently, regular polling incurs more overhead than our strategy.

Maintaining a Fair, Efficient, and Conservative Allocation

The Cilk-AP system uses algorithm DP to compute a fair, efficient, and conservative allocation in the first level of scheduling. The only information required for this calculation is the processor desire of each job, which is entered into the GAT by the job schedulers during the desire estimation process above. Since the GAT is stored in shared memory (in user space), it is not a requirement that the operating system perform the first level of scheduling. In Cilk-AP, the entries of the GAT are maintained in a distributed fashion by the job schedulers of the running jobs. In particular, each job scheduler recomputes the allocation when updating the processor desire of its own job: if the change in the job's desire does not affect the allocation, then no entries in the GAT are changed; otherwise, the job scheduler adjusts the allotments of the relevant jobs using algorithm DP. We assume that the number of jobs is always less than or equal to the number of processors in the system, so each job is allotted at least one processor. There are three primary reasons for maintaining the allocation in this way, related to the ease of implementation, timeliness, and overhead of our strategy. We explain these reasons below.

- *Ease of implementation.* Since the job scheduler already accesses the GAT to report the processor desire of its job, it does not take much more effort to update the job allotment entries in the event that the allocation has been disturbed. In addition, since processors are moved only one at a time by algorithm DP, it is possible for multiple job schedulers to update the GAT concurrently without a strict locking protocol; we explain this in more detail in Section 6.2.
- *Timeliness.* The allocation of the system only changes when a job enters or leaves the system, or when the current processor desire of a running job changes. Since the job scheduler access the GAT at precisely these times, it makes sense for the job scheduler to update the allocation as well. This way, we can be sure that changes to the allocation are always timely and necessary.
- *Overhead.* By updating the allocation only when needed, we minimize the overhead of maintaining the GAT while maximizing the responsiveness of the allocation to changes in job desires.

There are many alternatives to the above strategy for maintaining the GAT, most of which fall into one of two categories: strategies that store or update the GAT in kernel space, and strategies that store the GAT in user space but perform updates externally to all jobs (e.g., using a background processor). We summarize the differences between our strategy, which we call USER-JS (for "job scheduler"), and the

Strategy	Ease of implementation	Overhead	Security
USER-JS	easy	low but GAT could become a hot spot	job schedulers trust each other
USER-BP	easy	low	job schedulers trust background processor
KERNEL	difficult	high if job schedulers communicate with kernel	job schedulers trust kernel

Figure 6-3: Comparison of alternative strategies for maintaining the GAT.

alternative strategies, which we call KERNEL and USER-BP (for "background processor") respectively, in Figure 6.1. We use three main criteria for our comparison: ease of implementation, overhead, and security. We assume that the GAT is updated in a timely manner by all three strategies, as described in the timeliness condition above. (Alternatively, the GAT can be updated at regular intervals without coordinating with the job schedulers, reducing overhead at the cost of lower responsiveness.) From Figure 6.1, we see that the primary disadvantages of USER-JS are its lack of scalability (since the GAT may become a "hot spot") and the level of trust it requires between the job schedulers. We describe methods to cope with these problems in Section 6.2.

Adjusting a Job's Processor Usage

After setting a job's allotment in the GAT, the task still remains of adjusting the job's processor usage to match the computed allotment; the discrepancy between these two quantities is the difference between the terms p_j and a_j, as explained in Section 5.2. In the Cilk-AP system, adjustments to a job's processor usage are made by the job scheduler using sleep and wake signals, as specified by algorithm WSAP. For a given job j, let `signaled_to_sleep` be the number of working processors that have received a sleep signal from the job scheduler, but which have not yet gone to sleep. Every `est_cycle` seconds, after reporting j's desire to the GAT and making any necessary changes to the allocation, the job scheduler compares j's allotment a_j to its current usage p_j and takes one of two actions:

1. If $a_j < (p_j - \texttt{signaled\_to\_sleep})$, send a sleep signal to $(p_j - \texttt{signaled\_to\_sleep} - a_j)$ of the working processors that have not been signaled to sleep; if too few of those processors exist, send a sleep signal to all of them. Increment `signaled_to_sleep` for each sleep signal sent.
2. If $a_j > (p_j - \texttt{signaled\_to\_sleep})$, send a wake signal to $(p_j - \texttt{signaled\_to\_sleep} - a_j)$ processors that are asleep, waking all processors that have work first before waking those that do not have work. A processor that has been signaled to

sleep (but which hasn't gone to sleep yet) can be "woken up" by canceling its sleep signal and decrementing `signaled_to_sleep`.

Upon receiving a sleep signal, a processor that is working goes to sleep as soon as it can, decrementing `signaled_to_sleep` immediately prior to sleeping. (In our implementation, working processors can only go to sleep at the thread boundaries of a computation, as explained in Section 6.2.) A processor that is stealing checks to see if $a_j < p_j$ before each steal attempt; if it is, then the processor goes to sleep. If a stealing processor chooses a victim that is sleeping with a nonempty deque, then the processor wakes the victim up and goes to sleep itself (as specified by algorithm WSAP); the usage of j does not change in this case. We assume that j always has access to its allotted number of processors when increasing its usage in Action 2.

The strategy described above is identical to the strategy used by algorithm WSAP in the adversarial model (Section 4.1), with two important distinctions. First, since adjustments to the processor usage are not instantaneous in practice, we need to keep track of the number of processors that have been signaled to sleep. Second, sleep signals are only sent to processors that are busy working, because stealing processors go to sleep on their own. In Cilk-AP, the interval at which a job's processor usage is adjusted is the same as the desire-estimation interval, or `est_cycle` seconds. By using the same interval, we ensure that the frequency at which the job scheduler affects the system allocation is the same as the frequency at which it adjusts to changes in the allocation.

The overall reallocation process in Cilk-AP follows a coordinated policy (Chapter 2), because processors are only removed from a job at their earliest convenience. We discuss how this policy affects our implementation in Section 6.2. Since the job schedulers are responsible for both the first and second levels of scheduling, the problem of trust between the first and second-level schedulers now becomes a problem of trust between the job schedulers themselves. We explain our approach to this problem, again in Section 6.2.

Setting the Tunable Parameters

The job scheduler uses two tunable parameters to perform its second-level scheduling functions: `est_cycle` and η. In theory, both of these parameters can vary from job to job, but in Cilk-AP we only allow η to vary, and use the same value of `est_cycle` across all jobs. The motivation for this decision is to uphold the fairness and efficiency conditions of algorithm DP: since `est_cycle` is the interval at which a job's processor usage is adjusted, it also represents the responsiveness of the job to changes in the allocation, whether caused by the job itself or some other job in the system. If the value of `est_cycle` varies from job to job, then some jobs respond faster to allocation changes than others, resulting in potentially long periods of inefficiency or unfairness. For example, if job j has `est_cycle` $= x$ and job i has `est_cycle` $= 4x$, then a change in j's desire can take up to $3x$ seconds to be noticed by i, assuming that the intervals of i and j are aligned and that updates to the GAT are instantaneous. (If these assumptions are removed, then the delay may be even greater.) If j is increasing

its desire to demand more of its fair share of processors, then the allocation may become unfair during the delay, since i may be using some of the processors j needs. Conversely, if j is decreasing its desire below its fair share, then the allocation may become inefficient during the delay, since i may be able to use some of the processors j gives up. In practice, we cannot entirely avoid this type of delay, but we can minimize it by using the same value of `est_cycle` for all jobs.

Given that the value of `est_cycle` is fixed, we still must determine what value to use. Smaller values of `est_cycle` increase the job scheduler's responsiveness to changes in both the job's parallelism and the overall allocation, albeit at the cost of greater overhead. Larger values of `est_cycle` reduce the job scheduler's responsiveness to these changes, but incur less overhead. For our implementation, we tune the value of `est_cycle` experimentally, as described in Chapter 7.

Unlike `est_cycle`, the parameter η need not be consistent across all jobs, and can even be set by the user, because its effect on the allocation is safeguarded by algorithm DP. In particular, η is only used to compute the desire of a job, not its allotment. The job's allotment is computed in the first level of scheduling, and is guaranteed by algorithm DP never to exceed the job's fair share. We should note, however, that low values of η bloat the job's desire by allowing it to maintain a high ratio of stealing processors. In contrast, high values of η result in more conservative estimates, but may prevent the job from exploring its future parallelism. Setting η to 0.5 (50% efficiency) strikes a good balance between the two extremes, and is the value we used in most of our experiments in Chapter 7. In practice, it is possible to dynamically tune the value of η to respond to changes in the system load or the job itself. For example, if the system load is too high, then the job scheduler can increase η to reduce the number of processors used for stealing. Alternatively, if the system load is low, then the job scheduler can reduce η, because we can afford to be inefficient. The value of η can also be increased when the job enters a serial phase, and reduced when the job enters a phase of high (or unknown) parallelism. For simplicity, we do not dynamically tune the value of η in our current implementation, because using a fixed value made it easier to analyze our experiments in Chapter 7.

6.2 Implementation

We implemented the Cilk-AP system by extending the runtime system of Cilk, a language for multithreaded parallel programming based on ANSI C. Specifically, we extended Cilk's job scheduler to perform the three functions described in Section 6.1: estimating a job's processor desire, maintaining the allocation, and adjusting a job's usage to match its allotment. The resulting scheduler is called the Cilk-AP scheduler. We begin with a brief overview of the existing Cilk implementation and then describe the implementation of Cilk-AP.

The Cilk Scheduler

The existing Cilk scheduler uses algorithm WS to schedule the computation of a job (WS is described in Section 4.1). A Cilk job is a program execution consisting of a collection of Cilk ***procedures***; each processor in Cilk (called a ***worker***) maintains a deque of ready procedure instances. Cilk uses the THE protocol to manage the ready deque of each worker [19]. This protocol allows an exception to be signaled to a worker without introducing any additional work overhead. A worker checks for an exception every time it pops a frame from its deque.

When compiling a Cilk program, the program is first preprocessed to C using the `cilk2c` translator [34] and then compiled and linked with the Cilk runtime system for a target platform. The runtime system is responsible for scheduling the computation of a job (it contains the code for the Cilk scheduler) and is the only part of Cilk that we modified to implement Cilk-AP. The current implementation of Cilk has a portable runtime system that is designed to run efficiently on shared-memory symmetric multiprocessors (SMP's). Cilk runs on UNIX-like systems that support POSIX threads (Pthreads), which are used to implement the workers of a job. When a Cilk program is started, the user specifies the number of workers that the Cilk job can use; the runtime system then creates this many Pthreads and runs a worker on each thread. Cilk relies on the operating system to schedule the workers onto the physical processors of the machine.

The Cilk-AP Scheduler

The Cilk-AP system uses a technique called "process control" [48] to coordinate the allocation of processors between the first and second levels of scheduling. In the process control technique, jobs create virtual processors independently of the first-level scheduler. When the allotment of a job changes, the job is notified of the condition and is expected to adjust its current usage accordingly. Eventually, the total number of virtual processors becomes equal to the number P of physical processors in the system. Process control is consistent with the design of Cilk-AP, because of the way the job scheduler adjusts the usage of a job. It is also consistent with the existing implementation of Cilk, which virtualizes a job's processors using Pthreads. The extensions to the runtime system required to implement process control in Cilk-AP are therefore relatively simple.

To reduce the overhead of creating/destroying virtual processors, the Cilk-AP scheduler automatically creates P workers at the beginning of a job's execution. Meanwhile, the scheduler continues to run on its own Pthread, which we call the ***background thread***. To adjust the processor usage of a job, the workers are put to sleep or awakened according to the strategy specified in Section 6.1. The Cilk-AP scheduler creates an unnamed UNIX pipe for each worker to implement the sleeping mechanism: a worker goes to sleep by performing a blocking read on its pipe, and a sleeping worker is awakened when either the background thread or another worker writes to its pipe. Initially, the desire of a job is set to 1 processor, which causes most of the workers (which start out stealing) to put themselves to sleep. After this point,

the job's desire is measured periodically by the background thread using the process described in Section 6.1. The steal-attempt counters used to calculated `PUS_ratio` are maintained by each worker using the event-recording mechanism already built into Cilk.

According to the process-control technique, no worker is forcibly put to sleep by the Cilk-AP scheduler when reducing a job's usage. Instead, the scheduler signals a sleep exception to the worker from the background thread using the existing exception mechanism of Cilk [19]. Since the worker only checks for an exception after a frame is popped from its deque, it may take a while for the worker to notice the exception if the thread it is currently executing is very long (for example, if the worker is executing a subcomputation of the LOOPY benchmark). In the worst case, this delay causes the total number of workers in the system to exceed P, resulting in added overhead due to context switching and synchronization delays [17]. A similar situation occurs when the number of jobs in the system exceeds P, since each job is given at least 1 processor. While this phenomenon of process control temporarily affects the performance of running jobs, it is a much better alternative to policies that resort to queuing in these situations [52].

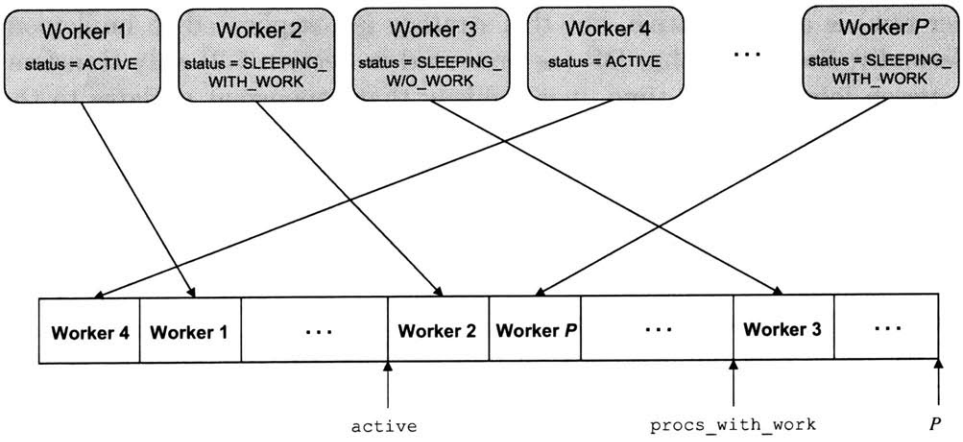


Figure 6-4: The thread state array (TSA) used by the Cilk-AP scheduler to organize the workers of a job. The first `active` workers are either working or stealing; the next `procs_with_work` − `active` workers are sleeping with work; and the last P − `procs_with_work` workers are sleeping without work.

To simplify the management of the workers, the Cilk-AP scheduler uses an array of indirection to group the workers according to their current status: active (the worker is either working or stealing), sleeping with work (the worker is asleep with a nonempty deque), and sleeping without work (the worker is asleep with an empty deque). Figure 6-4 illustrates this array of indirection, called the ***thread state array*** (TSA). The TSA simplifies a number of the operations performed by the Cilk-AP scheduler:

1. When a worker is stealing, it only chooses victims that are either active or

sleeping with work.

2. When the background thread increases the usage of a job, it only wakes up workers that are either sleeping with work or sleeping without work.
3. When the background thread reduces a job's usage, it only signals a sleep exception to workers that are active.

Accesses to the TSA by the different workers are synchronized with the use of Cilk locks [47]. Cilk locks support the memory-consistency semantics of release consistency [23, p. 716] and are designed to be as fast as is supported by the underlying hardware [47].

The final component of the Cilk-AP system is the GAT, which is currently stored in a memory-mapped file that all jobs have access to. Before creating the workers of a job, the Cilk-AP scheduler first maps the GAT file into memory and then registers the job (adding a row to the GAT) using the obtained pointer. The same pointer is used in subsequent updates to the GAT. Currently, we do not use a locking scheme to control accesses to the GAT for two main reasons. First, since every job accesses the GAT on a regular basis, using a coarse-grained lock on the table would make the GAT a bottleneck (or "hot spot") in the system. Using fine-grained locks (e.g., one lock per row) is an alternative, but this strategy is complicated to implement correctly. Secondly, since algorithm DP (as presented in Section 5.2) only transfers processors between jobs one at a time, it is unlikely that concurrent updates to the GAT can cause the allocation to deviate that much from the fair, efficient, and conservative ideal. If deviations do occur, the allocation can easily be rebalanced during the next update to the GAT. In general, we find that the cost of implementing an efficient locking scheme outweighs the overhead incurred by brief periods of inefficiency or unfairness in the allocation. Nevertheless, we consider more scalable approaches than the current scheme in Chapter 9.

By restricting our modifications to the Cilk runtime system, we automatically impose the condition that all jobs in the Cilk-AP system are Cilk jobs. Since each job is linked with the same (modified) runtime system, they are all associated with instances of the same job scheduler—the Cilk-AP scheduler. While this consistency limits the scope of our system, it eliminates the problem of trust between the job schedulers, because they all share the same code. In Chapter 9, we consider possible ways to generalize the Cilk-AP system to include other types of jobs as well.

Upholding the Design Goals of Cilk

The implementation of the Cilk-AP system upholds the three major design goals of the current Cilk implementation: simplicity, minimal work overhead, and portability. We summarize Cilk-AP's adherence to these goals below.

- *Simplicity.* The extensions made by Cilk-AP to the Cilk runtime system leverage existing mechanisms whenever possible, both simplifying and reducing the number of changes required. The process-control technique leverages Cilk's implementation of workers as Pthreads. The steal-attempt counters maintained

by each worker use the existing event-recording mechanism of Cilk. The background thread leverages Cilk's exception mechanism to signal workers to sleep. Finally, Cilk locks are used to control access to the TSA.

- *Minimal work overhead.* The Cilk-AP system uses steal-attempt information to estimate the desire of a job. Since this information is only gathered while a worker is stealing, it does not contribute to work overhead, thus satisfying the work-first principle of Cilk.
- *Portability.* The Cilk-AP scheduler performs both the first and second levels of scheduling and does not depend on the operating system or underlying architecture of the machine. The process-control technique escapes direct interaction with the underlying hardware by virtualizing the physical processors of the machine. The sleeping mechanism used to increase or decrease a job's usage is implemented using standard UNIX pipes. Finally, the GAT is implemented using the standard UNIX mechanism of memory-mapped I/O.

1

Chapter 7

Experimental Results

In this chapter, we present a suite of experiments that measure the overhead and performance of the Cilk-AP system. In each experiment, we compare Cilk-AP to the current Cilk system as follows. If there is only one job running, then we compare the results from the two systems directly. If there are multiple jobs running, then we compare Cilk-AP to the Cilk system combined with a static processor-allocation policy. (Recall that Cilk is only compatible with static allocation policies because it uses algorithm WS.) We begin by describing the different Cilk applications used in our experiments. Then, we describe our experiments for measuring the overhead of the Cilk-AP system, and use our results to choose an appropriate value for the tunable parameter `est_cycle`. Finally, we describe our experiments for analyzing the performance of Cilk-AP in the scenarios presented in Figure 2-1, and compare the execution times achieved by Cilk-AP to the greedy-scheduling bound proved in Theorem 3. Our results show that the Cilk-AP system incurs negligible overhead and provides up to 37% improvement in throughput and 30% improvement in response time for the tested scenarios.

All experiments in this chapter were performed on an idle SGI Origin 2000 SMP with 16 195-MHz processors and 8 Gb of memory, running version 6.5 of the SGI IRIX operating system.

Cilk Applications

We used several Cilk applications in our experiments. These applications are described below:

- `fib(n)` is a program that calculates the `n`th Fibonacci number using double recursion.
- `loopy(n)` is an implementation of the LOOPY benchmark from Section 4.2 that uses a single loop to spawn `n` equal-sized subcomputations. Each subcomputation consists of 10^6 iterations of a small amount of work.
- `cholesky(n,z)` is a program that performs a divide and conquer Cholesky factorization of a sparse symmetric positive-definite matrix. The input matrix

is generated randomly with size `n` and `z` nonzero elements; the nonzero elements are stored in a quad tree.

- `strassen(n)` is a program that multiplies two randomly generated $\mathtt{n} \times \mathtt{n}$ matrices using Strassen's algorithm. The program reverts to a simpler divide-and-conquer algorithm when the matrix size is sufficiently small (currently, less than 64).
- `knary(n,k,r)` is a synthetic benchmark whose parameters can be set to produce different values for work and critical path length. Written by Robert D. Blumofe of MIT's Computer Science and Artificial Intelligence Laboratory, `knary` generates a tree of depth `n` and branching factor `k`, where the first `r` children are executed serially and the remaining $\mathtt{k} - \mathtt{r}$ are executed in parallel. At each node of the tree, the program performs 100 iterations of a small amount of work.

All programs are compiled using `gcc -O2` when used in an experiment.

7.1 Overhead of Cilk-AP

In this section, we describe three experiments for measuring the overhead of the Cilk-AP system. In the first experiment, we ran single jobs with high parallelism in isolation and measured the amount of work performed before all of the job's processors get work on their deques; we call this amount the ***all-procs*** time. The second experiment is identical to the first, except we measured the overall running time of the job instead. By performing these experiments for different values of `est_cycle`, we can analyze the trade-off between the all-procs time and running time of each job, and then use this information to choose an appropriate value for `est_cycle`; this value is used in all subsequent experiments. In the third overhead experiment, we performed a simple test to illustrate the overhead incurred by process control when the total number of workers in the system exceeds P.

The target efficiency η of the Cilk-AP scheduler was set to 0.5 in all three experiments. For accuracy, each data point reported (either an all-procs time or a running time) was averaged over 5 trials.

7.1.1 Experiment O1: All-Procs Time Overhead

The first overhead experiment measures the all-procs time of several jobs, each of which has parallelism much greater than P. The jobs were run in isolation with all P processors using both the Cilk and Cilk-AP systems. We first measured the parallelism T_1/T_∞ of each job using the built-in profiling mechanism available in Cilk; the profiler was not used during the actual P-processor runs. Since all of the applications we used are deterministic—meaning that the computation only depends on the program and its inputs—we could measure the work T_1 by directly timing the 1-processor run. The critical path length T_∞ was measured internally by the Cilk profiler. Since the profiler incurs a significant amount of overhead, the measured values of T_1 predict a much slower running time on P processors than our results indicate (where profiling is turned off). We can ignore this discrepancy, however,

Cilk Job	T_1	T_∞	T_1/T_∞	All-procs time (16 procs.)	Running time (T_{16})
`fib(33)`	189.9	0.001093	169152	0.0709	1.044
`loopy(64)`	279.2	4.366	63.95	0.1089	18.15
`cholesky(2048,10000)`	373.8	0.4843	771.7	0.1271	11.17
`knary(10,6,1)`	1386	0.3748	3698	0.1199	12.89

Figure 7-1: The parallelism of four Cilk jobs, derived from work and critical path measurements. The all-procs time and running time of each job using the Cilk system is shown on the right. All times are in seconds.

because we are only using the profiler to measure the parallelism of a job, which is the *ratio* between T_1 and T_∞, under the assumption that profiling affects T_1 and T_∞ comparatively.

Figure 7.1.1 shows the work, critical path length, and parallelism of four different jobs; each job has parallelism much greater than the number of processors in our system, $P = 16$. The all-procs time of each job using the Cilk system (on 16 processors) is also shown in Figure 7.1.1. Using this value as a basis for comparison, we then measured the all-procs time of each job using the Cilk-AP system for different values of `est_cycle`. We started with `est_cycle` = 1 ms and repeat our measurements for `est_cycle` = 2, 5, 10, 25, 50, and 100 ms. (The resolution of the hardware clocks on our machine is approximately 0.8 ms.) The measured all-procs times are shown in Figure 7-2(a). Figure 7-2(b) plots the same results in terms of the number of `est_cycle` intervals.

Since a job starts out with only 1 processor in Cilk-AP, we expect the all-procs times reported in Figure 7-2(a) to be longer than the times recorded for Cilk, where the job starts out using all P processors. We compare the two systems in Figure 7-3. A value of 1 in Figure 7-3 indicates that the all-procs time using Cilk-AP is as short or shorter than when using Cilk, for the given job and value of `est_cycle`.

In Cilk-AP, the desire of a job grows by at most a factor of $1/\eta$ during each `est_cycle` interval, so it takes at least $\log_{1/\eta} P$ intervals before a job requests all P processors of the machine. Once a processor has been requested and allocated to the job, there is a short delay before the job's usage actually increases, since the Cilk-AP scheduler must wake up a sleeping worker, and another delay before the worker gets work on its deque, since it has to steal this work from someone else. The total τ of these delays over all P workers represents the all-procs time of the job when `est_cycle` = 0. Assuming each delay takes a constant amount of time, τ can be expressed as follows:

$$\tau = O(P) + O(\log_{1/\eta} P) \ . \tag{7.1}$$

The first term represents the time taken to wake up all P workers, performed (in series) by the Cilk-AP scheduler from the background thread; and the second term represents the time for all P workers to find work, performed (in parallel) by the awakened workers during each `est_cycle` interval. In practice, `est_cycle` is greater

than 0, so τ is distributed across several intervals. If `est_cycle` is large enough, then each portion of τ completes within an interval, and the all-procs time of the job is bounded by $\log_{1/\eta}$ intervals. If `est_cycle` is small, then the all-procs time may take longer than $\log_{1/\eta}$ intervals. For the values of `est_cycle` used in our experiment, the all-procs time is always much longer than $\log_{1/\eta} P = \log_2 16 = 4$ intervals, as shown in Figure 7-2(b). The number of intervals drops rapidly, however, as `est_cycle` is increased. When reading Figure 7-2(b), observe that while the number of intervals decreases for larger values of `est_cycle`, the value of the all-procs time actually increases, as shown in Figure 7-2(a).

Figure 7-2 illustrates the trade-off between overhead and responsiveness in the Cilk-AP system. The smaller the value of `est_cycle`, the shorter the all-procs time, but the greater the number of desire-estimation intervals performed by the Cilk-AP scheduler. In other words, while Figure 7-2(a) tells us to use the smallest value of `est_cycle` possible, Figure 7-2(b) cautions us from using a value that is too small, lest it incur significant overhead. Looking at both graphs, it seems that a value between 5 and 10 ms strikes a good balance between the two extremes. We investigate this trade-off further in the next experiment, and use our combined results to choose an appropriate value for `est_cycle`.

As an aside, we compared the all-procs time of the `loopy` program using Cilk-AP to the all-procs time measured using Cilk-AP-INST, a version of Cilk-AP that estimates the processor desire of a job by directly measuring its instantaneous parallelism. In Cilk-AP-INST, the instantaneous parallelism of a job is measured by counting the total number of threads on the ready deques of all workers. As explained in Section 4.2, policies of this nature are inherently misleading because they can substantially underestimate the actual parallelism of a computation. The results of our comparison support this claim, and are shown in Figure 7-4. As predicted, the Cilk-AP-INST system performs considerably worse than Cilk-AP for all values of `est_cycle`; even at the smallest interval size, Cilk-AP-INST is over 2.5 times slower than Cilk-AP.

7.1.2 Experiment O2: Running Time Overhead

In Experiment O1, we observed a trade-off between the overhead and responsiveness of Cilk-AP as the value of `est_cycle` is changed. In Experiment O2, we assess the trade-off further by measuring its effect on the running time of each job. For consistency, we used the same jobs listed in Figure 7.1.1 from the first experiment. We then measured the running time of each job using both the Cilk and Cilk-AP systems over the same range of values for `est_cycle`. By comparing the running times of Cilk-AP to Cilk, we can gauge the overhead incurred by the desire-estimation process, which is performed by the Cilk-AP scheduler during every interval. For small values of `est_cycle`, this process occurs with high frequency throughout the job's execution, and hence we expect the overhead to be high. As `est_cycle` is increased, the process occurs less frequently, and the resulting overhead is low. Figure 7-5 shows our results for the different values of `est_cycle`. The overhead of Cilk-AP is expressed as a percentage of the running time using Cilk; a value of 0 indicates that Cilk-AP's running time is

as short or shorter than the running time using Cilk.

From Figure 7-5, we see that the overhead of Cilk-AP drops rapidly as the value of `est_cycle` is increased. In particular, for `est_cycle` > 5 ms, the overhead is virtually nonexistent. The decline of the overhead in Figure 7-5 is consistent with the decline of the number of intervals in Figure 7-2(b). In particular, both graphs suggest that a value of `est_cycle` greater than 5 ms is large enough to avoid significant overheads when using Cilk-AP. Beyond 5 ms, the difference in overhead is minimal. Combining this result with our analysis of Figure 7-2(a), we conclude that a value `est_cycle` $= 5$ ms is an appropriate choice for our system.

By repeating the first and second experiments for different values of P, we can make a general statement about the overhead and optimal interval size of Cilk-AP in terms of the number of processors in a system. We describe a strategy for performing this extrapolation in Chapter 9.

7.1.3 Experiment O3: Process Control Overhead

In this experiment, we performed a simple test to measure the overhead incurred by process control when the number of workers in the system exceeds P. As observed in Section 6.2, this situation usually occurs when the threads of a computation are very long, because a worker that is signaled to sleep only notices the exception after completing the currently executing thread. To simulate this effect, we ran two instances of the `loopy` program—each with parallelism greater than or equal to P—separated by the all-procs time of the first instance. In other words, we allow the first job's usage to reach P processors before starting the second job. In the Cilk-AP system, the allotment of the first job is reduced to $P/2$ processors when the second job arrives, but there is a potentially long delay before the first job is actually able to reduce its current usage, because each of its workers is busy executing a long subcomputation. The second job is oblivious to this delay and begins using its fair share of $P/2$ processors immediately after starting. As a result, the total number of workers in the system exceeds P for as long as the first job is unable to properly reduce its usage.

The ***response time*** of a job is defined as the time elapsed from when the job arrives to when it completes (including any time the job spends queued). We measured the mean response time of the two jobs for different input sizes, once using Cilk-AP and again using Cilk-AP with an allocation limit of $P/2$ processors per job. In the second case, the total number of workers never exceeds P, because the first job is limited to $P/2$ processors from the start. Our results are shown in Figure 7.1.3. For comparison, we ran the experiment using the Cilk system with a static allocation of $P/2$ processors per job. From Figure 7.1.3, one can see that the process control technique incurs significant overhead relative to Cilk when the number of workers in the system exceeds P. If the number of workers is kept at or below P, however, process control incurs no additional overhead, as evidenced by the last column of Figure 7.1.3. The larger the input size to `loopy`, the lower the process control overhead, because the first job has more time to reduce its processor usage (in fact, all subcomputations after the first P are executed using only $P/2$ processors by the first job). The allocation limits used in this experiment are for illustrative purposes only, and are

not part of the design or implementation of the Cilk-AP system. Even though such limits help prevent the number of workers from exceeding P, they also waste system resources and limit achievable utilization, as explained in Chapter 2.

7.2 Performance of Cilk-AP

In this section, we describe three experiments for measuring the performance of the Cilk-AP system. In the first experiment, we simulated scenario (a) from Figure 2-1 using three identical instances of the `fib` program. In the second experiment, we simulated scenario (b) from Figure 2-1 using complementary runs of the `knary` program. We compare the mean response time and throughput achieved by the Cilk and Cilk-AP systems in both of these experiments. We use the notion of ***power***, defined as the throughput divided by the response time [25, 26], to compare the overall performance. Power is a suitable metric for comparison because it reflects our goal of maximizing the throughput while minimizing the response time. In the last experiment, we measured the running times of several jobs using Cilk-AP and compare them to the greedy-scheduling bound proved in Theorem 3.

The target efficiency η of the Cilk-AP scheduler was set to 0.5 in the first two experiments and 0.75 in the third. For accuracy, all of the measured running times were averaged over 5 trials.

7.2.1 Experiment P1: Arrival of a New Job

In this experiment, we simulated scenario (a) from Figure 2-1 using three identical jobs, called A, B, and C, that request $P/2$ processors each. In the given scenario, jobs A and B start running at time 0 and job C arrives at some time $t \geq 0$. We begin by analyzing the mean response time and throughput achieved by the Cilk and Cilk-AP systems theoretically, assuming an ideal machine with no scheduling overhead. If T is the execution time of each job on $P/2$ processors, then the mean response time and throughput of each system can be expressed using P, T, and t. We calculated these values for $t = 0$ and $t = T/2$ and list them in Figure 7.2.1. For both values of t, the mean response time using Cilk-AP is longer than when using Cilk, but the increase in throughput is still large enough to yield a greater power for Cilk-AP. The ratios in Figure 7.2.1 indicate the relative performance we can expect from Cilk-AP and Cilk in practice.

In our experiment, we used three instances of `fib(38)` to represent jobs A, B, and C. Since each job has parallelism much greater than P, we set an allocation limit of $P/2$ processors to simulate scenario (a) correctly. Figure 7.2.1 shows the mean response time, throughput, and power achieved by the Cilk and Cilk-AP systems during the actual experiment. The value of T was measured experimentally for each system. Given the nonideal conditions of our machine, the ratios in Figure 7.2.1 coincide reasonably well with the theoretical values shown in Figure 7.2.1. As predicted, the power achieved by Cilk-AP is greater than the power of Cilk for both values of t, even with a theoretical ratio as small as 1.067 (as predicted for $t = T/2$).

7.2.2 Experiment P2: Changes in Parallelism During Runtime

In the second performance experiment, we simulated scenario (b) from Figure 2-1 by running two jobs, called A and B, that have complementary parallelism profiles. We divided each job into two phases, a serial phase and a parallel phase, and used the `knary` program to implement each phase. Job A executes the serial phase before executing the parallel phase, and job B executes the parallel phase before executing the serial phase. The pseudocode for each job is shown below. We used `knary(11,4,4)` for the serial phase and `knary(11,5,0)` for the parallel phase.

```
Job A:

cilk int main (...) {
  ...
  /* serial phase */
  spawn knary(11,4,4);
  /* wait for serial phase
     to complete */
  sync;
  /* parallel phase */
  spawn knary(11,5,0);
  sync;

  return 0;
}
```

```
Job B:

cilk int main (...) {
  ...
  /* parallel phase */
  spawn knary(11,5,0);
  /* wait for parallel phase
     to complete */
  sync;
  /* serial phase */
  spawn knary(11,4,4);
  sync;

  return 0;
}
```

Since A and B are perfect complements of each other, they exhibit the same amount of parallelism. We measured this value to be 13.68 using the Cilk profiler. To perform our experiment, we ran jobs A and B concurrently using both the Cilk and Cilk-AP systems and measured the mean response time in each case. Our results are shown in Figure 7.2.2. When using the Cilk system, each job is given the maximum allotment of $P/2 = 8$ processors, based on their reported parallelism (which exceeds this value). Since the allotment of each job is static, it remains fixed throughout both the serial and parallel phases of execution. In the Cilk-AP system, the allotment of a job is determined dynamically by the Cilk-AP scheduler based on estimates of the job's current desire. As a result, the allotment can grow and shrink during runtime to adapt to changes in the job's parallelism. In the first half of our experiment, job A executes the serial phase while job B executes the parallel phase, which causes the Cilk-AP scheduler to allocate most of the processors to job B. In the second half of our experiment, the situation is reversed, and job A receives most of the processors. Since Cilk-AP is able to use the processors more efficiently than Cilk, it achieves a shorter mean response time in Figure 7.2.2, completing both jobs about 30% faster than Cilk. Figure 7.2.2 also shows the value of $\overline{P}$ (defined in Section 3.2) for each job

when using Cilk-AP. We approximate $\overline{P}$ by measuring the average processor usage of each job over all `est_cycle` intervals. In practice, the parallel phase of our experiment takes longer than the serial phase (even when all 16 processors are used), and so the value of $\overline{P}$ tends to be greater than 8.

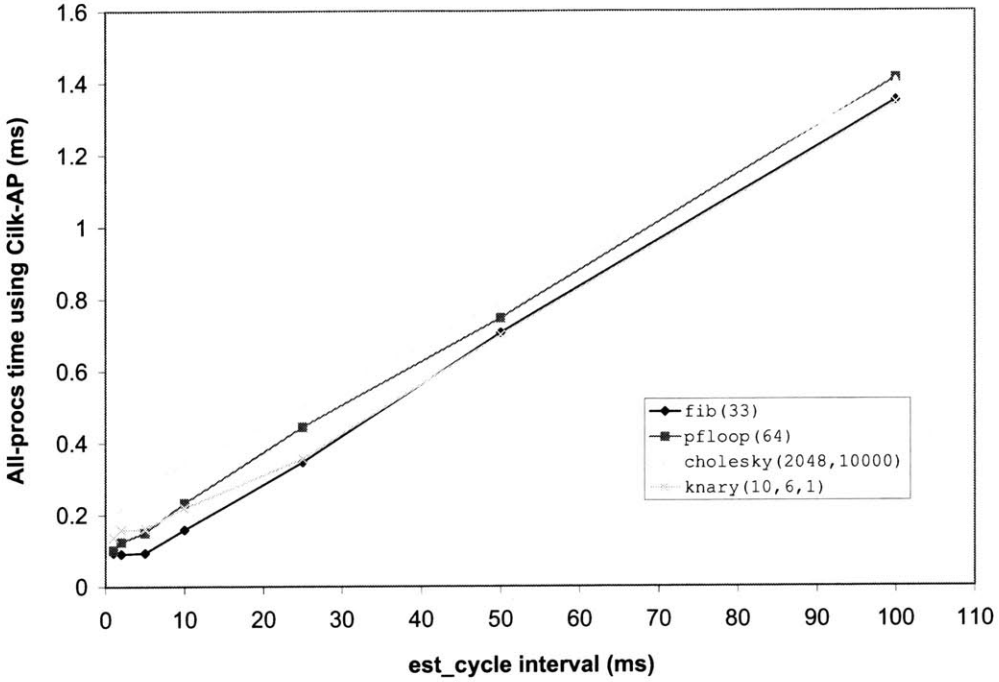
7.2.3 Experiment P3: The Greedy-Scheduling Bound

In this experiment, we measured the running times of different jobs using the Cilk-AP system and compare these times to the greedy-scheduling bound from Theorem 3. Our goal is to show that the Cilk-AP scheduler achieves asymptotically optimal time in practice, which suggests that a theoretical analysis of algorithm WSAP should yield the same result (we explore this analysis in future work). Since the greedy-scheduling bound becomes tighter with larger $\overline{P}$, we raised the target efficiency η to 0.75 to increase the accuracy and economy of our processor desire estimates. A higher value of η means that fewer processors are used inefficiently by a job, resulting in a lower value of $\overline{P}$.

Given an adaptively parallel job with work T_1, critical path length T_∞, and average processor allotment $\overline{P}$, a greedy scheduler executes the job in time $T \leq T_1/\overline{P} + T_\infty$, as proved in Theorem 3. In Figure 7.2.3, we compare the running times of different jobs using the Cilk-AP system to the greedy-scheduling bound predicted in Theorem 3. As before, we approximate $\overline{P}$ using the average processor usage of a job over all `est_cycle` intervals. (We use the job's usage instead of its allotment because the two quantities are not always the same.) In every case, the running time T using Cilk-AP is within a factor of 2 of the greedy-scheduling bound, and the performance of Cilk-AP closely matches that of Cilk on a fixed allocation of $\lceil \overline{P} \rceil$ processors. Since the Cilk scheduler achieves asymptotically optimal time in the static case (when the job's allotment is fixed) [4, 7], these results suggest that a similar bound can be proved for the Cilk-AP scheduler in the dynamic case (when the allotment is allowed to vary).

The running times for Cilk shown in Figure 7.2.3 come remarkably close to the best running time achievable by Cilk on *any* number of processors. This observation suggests that the Cilk-AP scheduler is able to find the optimal allotment of a job using the desire-estimation strategy presented in Section 6.1. We expect the accuracy of $\overline{P}$ to improve even further as the value of η is increased.

a)



b)

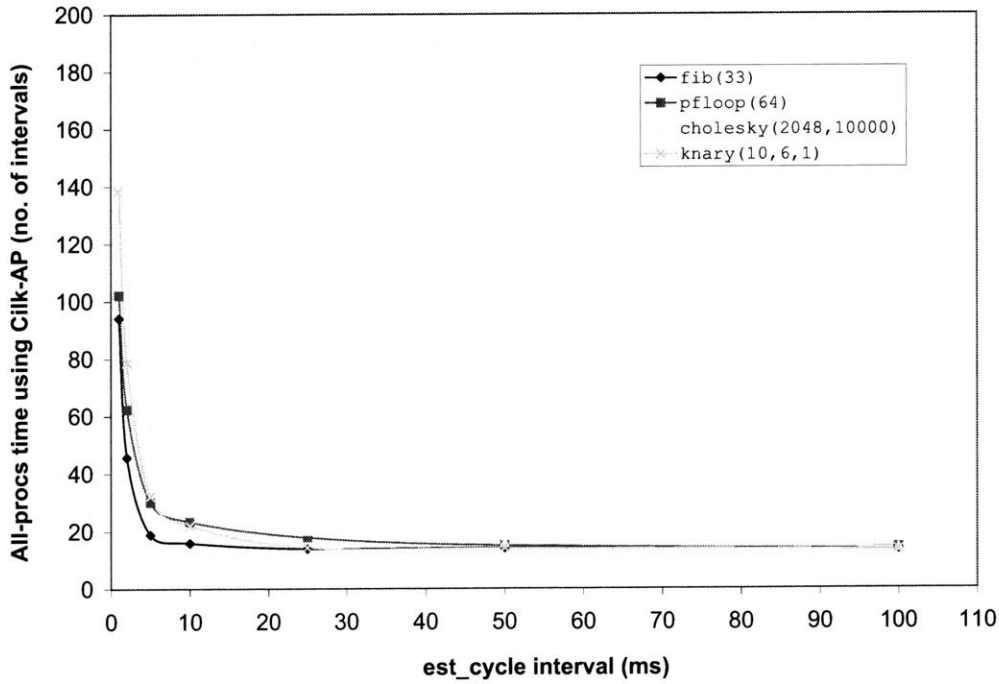


Figure 7-2: The all-procs time of different jobs using the Cilk-AP system, shown on an absolute scale in (a) and in terms of `est_cycle` in (b).

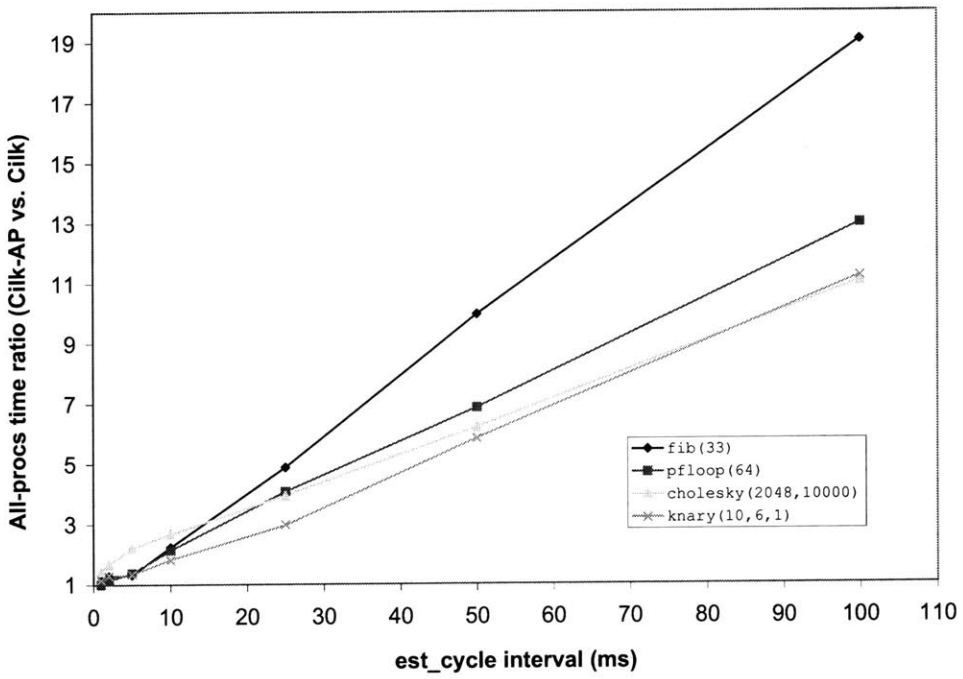


Figure 7-3: The all-procs time of different jobs using the Cilk-AP system, shown as a ratio of the all-procs time using Cilk.

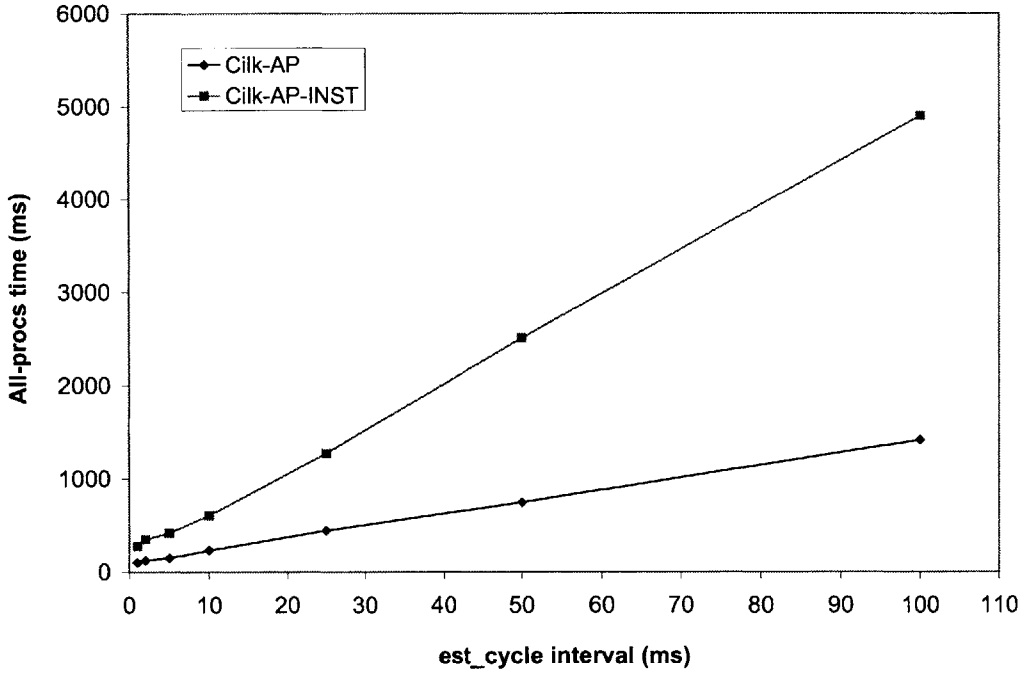


Figure 7-4: The all-procs time of the `loopy` program using Cilk-AP and Cilk-AP-INST, a version of Cilk-AP that uses instantaneous parallelism measurements to estimate a job's desire.

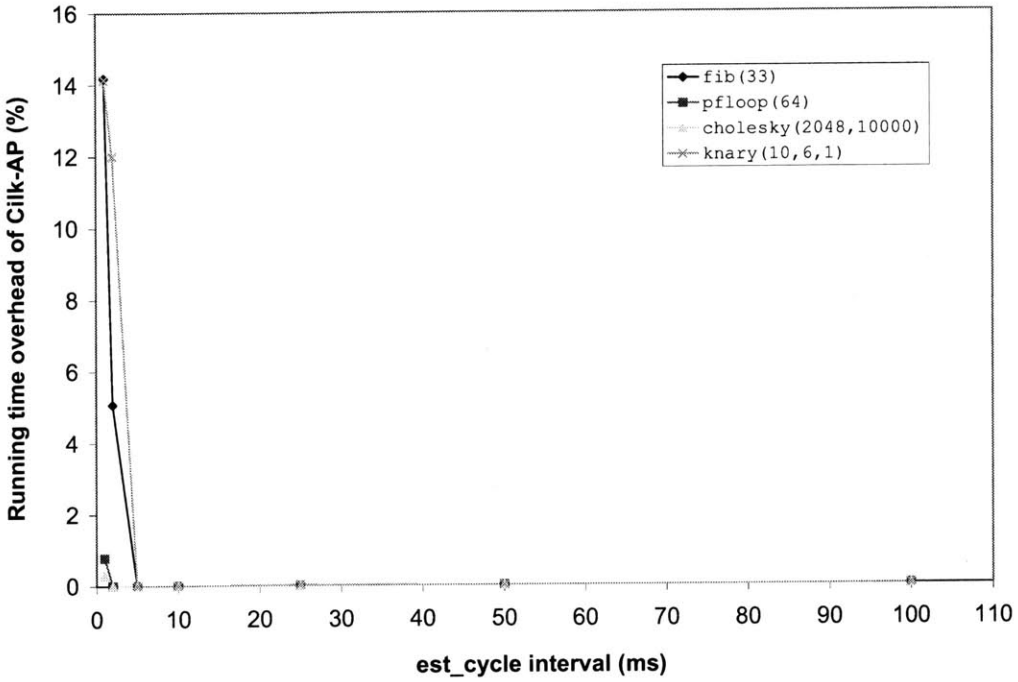


Figure 7-5: The running time overhead of the Cilk-AP system for different jobs, expressed as a percentage of the corresponding running time using Cilk.

Job 1	Job 2	Cilk (w/ alloc. limit)	Cilk-AP	Overhead (%)	Cilk-AP (w/ alloc. limit)	Overhead (%)
`loopy(16)`	`loopy(16)`	9.035	12.31	36.26	8.984	0
`loopy(32)`	`loopy(32)`	17.83	19.70	10.47	17.83	0.01528
`loopy(64)`	`loopy(64)`	36.21	37.68	4.078	35.81	0

Figure 7-6: The overhead of process control when the number of workers in the system exceeds P (= 16 in the above experiments). Each time represents the mean response time of jobs 1 and 2 in seconds.

Metric	$t = 0$			$t = T/2$		
	Cilk	Cilk-AP	Ratio	Cilk	Cilk-AP	Ratio
Mean response time (sec)	$4T/3$	$3T/2$	1.125	$7T/6$	$5T/4$	1.071
Throughput (jobs/sec)	$3/2T$	$2/T$	1.333	$3/2T$	$12/7T$	1.143
Power (jobs/sec^2)	$9/8T^2$	$4/3T^2$	1.185	$9/7T^2$	$48/35T^2$	1.067

Figure 7-7: Theoretical values for the mean response time, throughput, and power achieved by the Cilk and Cilk-AP systems in scenario (a) of Figure 2-1. The ratio of Cilk-AP to Cilk is shown for each metric.

Metric	$t = 0$			$t = T/2$		
	Cilk	Cilk-AP	Ratio	Cilk	Cilk-AP	Ratio
Mean response time (sec)	21.10	23.26	1.102	18.48	19.76	1.069
Throughput (jobs/sec)	0.09450	0.1291	1.366	0.09450	0.1022	1.082
Power ($\times 10^{-3}$ jobs/sec^2)	4.478	5.550	1.239	5.115	5.175	1.012

Figure 7-8: Experimental values for the mean response time, throughput, and power achieved by the Cilk and Cilk-AP systems in scenario (a) of Figure 2-1. The ratio of Cilk-AP to Cilk is shown for each metric. Three instances of `fib(38)` are used to represent jobs A, B, and C.

Metric	Cilk	Cilk-AP
Mean response time (sec)	31.33	22.12
$\overline{P}$ (Job A, Job B)	(8, 8)	(8.275, 9.756)

Figure 7-9: The mean response time and processor usage of the Cilk and Cilk-AP systems in scenario (b) of Figure 2-1. The `knary` program is used to implement the serial and parallel phases of jobs A and B.

Job	T_1	T_∞	$\overline{P}$	$T_1/\overline{P} + T_\infty$	Cilk-AP (T)	Cilk ($T_{\lceil \overline{P} \rceil}$)
`knary(11,3,3)`	7.669	4.187	3.732	6.242	11.37	10.85
`knary(11,4,0)`	122.6	0.001212	15.02	8.164	8.704	8.166
`knary(10,4,2)`	35.60	1.442	11.49	4.540	8.617	8.580
`strassen(1024)`	10.77	2.582	6.784	4.170	3.450	3.906
`pfloop(64)`	279.2	4.366	15.82	22.01	18.55	18.49
`fib(33)`	208.2	0.00704	15.60	13.35	18.99	18.93
`cholesky(2048,10000)`	448.4	0.8224	14.45	31.85	43.74	45.49

Figure 7-10: The running times of different jobs using the Cilk-AP system, compared here to the greedy-scheduling bound in Theorem 3. All times are in seconds.

Chapter 8

Related Work

The problem of scheduling adaptively parallel jobs on multiprogrammed parallel systems has been studied extensively in the past. In this chapter, we highlight some of this work in the context of the Cilk-AP system. For a thorough treatment of job scheduling in multiprogrammed parallel systems, the reader is directed to Feitelson's survey [17]. A more recent report has also been written by Feitelson and Rudolph [18]. For background on the concept of work stealing, the reader is referred to Blumofe's PhD thesis [4].

Dynamic processor-allocation systems are specifically designed with adaptively parallel jobs in mind. The term "adaptive parallelism" itself seems to have been coined by the designers of Piranha [11, 12], a dynamic processor allocation system based on the Linda programming model [13]. Piranha is considered to be one of the first real implementations of dynamic partitioning on a parallel machine [17]. Most dynamic allocation systems in the past have used the instantaneous parallelism of a job—usually determined by the number of threads that are ready to execute—to estimate the job's processor desire. We have shown in Section 4.2 that this approach is inherently inaccurate for a large class of parallel programs. Nguyen et. al. use runtime measurements of efficiency and/or speedup to tune a job's allotment [36,37], while others study the effect of using various application characteristics to influence allocation decisions [9, 14, 27, 29, 42, 44, 45]. These systems are either impractical, because they assume that application characteristics are provided to the scheduler beforehand, or based solely on recent measures of a job's performance. In contrast, the Cilk-AP system uses a policy that proactively explores the future parallelism of a job, and requires no *a priori* information about the job.

For the specific case of work-stealing jobs, little work has been done to analyzing their performance in multiprogrammed environments. The work-stealing algorithm presented by Blumofe and Leiserson [7], algorithm WS, is simultaneously efficient with respect to time, space, and communication, but assumes that the number of processors used to execute a multithreaded computation is fixed. As a result, any implementations of this algorithm, such as the Cilk system [5, 19, 47], can only be used with a static processor-allocation policy in a two-level scheduling environment. In more recent work, Arora et. al. [2] present a nonblocking implementation of algorithm WS that runs efficiently on a variable number of processors, making it suitable for

use with a dynamic allocation policy. Given a multithreaded computation with work T_1 and critical-path length T_∞, and for any number P of processes, their nonblocking work stealer executes the computation in expected time $O(T_1/P_A + T_\infty P/P_A)$, where P_A is the average number of processors allocated to the computation. The difference between this algorithm and WSAP is that the number of processes in WSAP always matches the number of physical processors allocated to the computation. We investigate the performance of WSAP and the Cilk-AP system in future work.

Part of the work presented in this thesis is based on previous (unpublished) work by Bin Song, Robert Blumofe, and Charles Leiserson of the Supercomputing Technologies Group at MIT. Song et. al. show that if a Cilk job running on P processors spends a substantial fraction of its time stealing, then P is bigger than the average parallelism of the computation. If the job only spends a small fraction of its time stealing, then P is smaller than the average parallelism. This claim is justified both theoretically and empirically in their work, and forms part of the intuition behind using steal-attempts statistics in the desire-estimation process of Cilk-AP.

Chapter 9

Conclusion

This thesis has presented the theoretical foundation, design and implementation of a two-level scheduling system for adaptively parallel work-stealing jobs. The Cilk-AP system uses dynamic partitioning (algorithm DP) in the first level of scheduling to allocate processors to jobs in a fair, efficient, and conservative manner. In the second level of scheduling, Cilk-AP uses a randomized work-stealing algorithm (algorithm WSAP) to schedule a job's computation, subject to these changing allocations. The Cilk-AP scheduler estimates the processor desire of a job using a policy that proactively explores the job's "future" parallelism. We have shown through the LOOPY benchmark that policies that directly measure the instantaneous parallelism of a job are inherently inaccurate. These policies grossly underestimate the average parallelism of a job for a large class of programs. The information used by the Cilk-AP scheduler to estimate a job's desire is provided by algorithm WSAP at virtually no additional cost. By relying solely on steal-attempt statistics, we guarantee that the overhead incurred by the desire estimation process is borne by the critical path and proportional to the inefficiency of the job, thus satisfying the work-first principle.

As a consequence of our design choices, the implementation of the Cilk-AP system requires only a few extensions to the current implementation of Cilk. Cilk-AP uses the process control technique to manage the global allocation in user space. By leveraging existing mechanisms of Cilk and relying only on standard UNIX mechanisms, the Cilk-AP system upholds the three major design goals of the Cilk implementation: simplicity, minimal work overhead, and portability. At present, all jobs in the Cilk-AP system are Cilk jobs, which simplifies issues like security and trust between the job schedulers, at the cost of limiting the system's scope. We consider ways to generalize the Cilk-AP system, and other extensions to the current design, in our discussion of future work below.

Future Work

This thesis is the result of ongoing research by the Supercomputing Technologies Group at MIT. As such, parts of the work presented here are pending further investigation and study. In this section, we discuss these areas and directions for future work

in the context of the Cilk-AP system. We begin with areas pertaining to the first level of scheduling, and then discuss areas pertaining to the second level. Finally, we describe some experiments that can be done to extend (and strengthen) our results.

The Cilk-AP system uses algorithm WSAP to schedule a job's computation in the second level of scheduling. We have described WSAP in the context of an adversarial model, but have not provided a theoretical analysis of its performance; this analysis is part of our current research. Like algorithm WS, our goal is to show that WSAP achieves provably good time, space, and communication bounds. The nature of these bounds is unclear to us at this time.

We are also exploring different extensions and improvements to the desire estimation process of Cilk-AP. One idea is to incorporate the history of a job's processor usage when estimating its current desire, giving more weight to past usage trends than to current demands. This technique is particularly useful for moderating the effect of short-lived bursts or drops in a job's parallelism. Another idea is to implement a mechanism for detecting whether or not a worker is blocked on I/O, similar to the mechanism used by McCann et. al. in their dynamic processor allocator [33]. If a worker of a job is blocked, then it can be safely discounted from the job's current usage.

The Cilk-AP system uses algorithm DP to allocate processors to jobs in the first level of scheduling. DP uses a strict notion of fairness that does not distinguish between jobs. In practice, it may be useful to implement a priority scheme for distinguishing between different jobs (or users of jobs). A job's priority can be set by the user, and the user's priority can be set by the administrator of the machine, for example. Alternatively, the priority of the job can be determined by the Cilk-AP system itself, based on runtime characteristics or properties of the job. The dynamic processor allocator designed by McCann et. al. [33] uses an adaptive priority mechanism that increases the priority of jobs using fewer processors than the system equipartition, and lowers the priority of jobs using more than the system equipartition. The allocator uses a job's priority to determine which processors it can preempt on the job's behalf—for example, a lower priority job can never cause the preemption of a processor from a higher priority job. Once a priority scheme is established for Cilk-AP, the first-level scheduler can be modified in a similar manner to use this information when reallocating processors between jobs.

Instead of implementing its own priority scheme, the Cilk-AP system can leverage existing priority mechanisms if we move the first-level scheduler into kernel space. Although this strategy reduces the portability of the system, it extends the scope of Cilk-AP to all types of jobs, not just Cilk jobs. As we noted in Section 6.1, the disadvantages of a kernel-level implementation include the communication overhead between the first and second levels of scheduling and the issue of trust between the two levels. We are currently exploring these issues, as well as other ways to generalize the Cilk-AP system.

A more immediate problem in the first level of scheduling is the potential for the GAT to become a hot spot, especially on systems with a large number of processors. One solution to this problem is to use a separate, dedicated processor to periodically update the allotments in the GAT, relieving the individual job schedulers of this

responsibility. If the updates are made frequently enough, we expect the allocation to be as responsive to changes in job desires as the current system is (at the cost of dedicating a separate processor for this role).

Finally, there are several experiments that we are planning to do to extend our results from Chapter 7. First, we are considering an alternative way to measure the overhead of the Cilk-AP system, using the amount of work performed by a job before all processors get work on their dequeus (instead of measuring the time this process takes). By repeating this experiment for different values of P, we can make a general statement about the startup overhead of our system in terms of the size of the machine and the value of `est_cycle`. This overhead determines the minimum job size for which it is advantageous to use our system. A similar statement can be made about the optimal value of `est_cycle` in terms of the machine size, by repeating the second overhead experiment for different values of P. Together, these results can make a strong case for the scalability and performance of Cilk-AP on any machine size. As a final note, we plan to test the Cilk-AP system on real-world applications, such as protein folding and ray tracing, to further reinforce these results.

Bibliography

[1] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[2] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the 10th ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, Puerto Vallarta, Mexico, June 1998.

[3] Michael A. Bender and Michael O. Rabin. Scheduling Cilk multithreaded computations on processors of different speeds. In *Proceedings of the 12th ACM Symposium on Parallel Algorithms and Architectures*, pages 13–21, Bar Harbor, Maine, United States, July 2000.

[4] Robert D. Blumofe. *Executing Multithreaded Programs Efficiently.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 1995.

[5] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 25 1996.

[6] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. *SIAM Journal on Computing*, 27(1):202–229, February 1998.

[7] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, September 1999.

[8] Timothy B. Brecht. An experimental evaluation of processor pool-based scheduling for shared-memory NUMA multiprocessors. In Dror G. Feitelson and Larry Rudolph, editors, *Proceedings of the 3rd Workshop on Job Scheduling Strategies for Parallel Processing*, pages 139–165, Geneva, Switzerland, April 1997. Springer-Verlag. Lecture Notes in Computer Science Vol. 1291.

[9] Timothy B. Brecht and Kaushik Guha. Using parallel program characteristics in dynamic processor allocation policies. *Performance Evaluation*, 27 & 28:519–539, October 1996.

[10] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *Journal of the ACM*, 21(2):201–206, April 1974.

[11] Nicholas Carriero, Eric Freeman, and David Gelernter. Adaptive parallelism on multiprocessors: Preliminary experience with Piranha on the CM-5. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the 6th Annual Workshop on Languages and Compilers for Parallel Computing*, pages 139–151, Portland, Oregon, United States, August 1993. Springer-Verlag.

[12] Nicholas Carriero, Eric Freeman, David Gelernter, and David Kaminsky. Adaptive parallelism and Piranha. *IEEE Computer*, 28(1):40–49, January 1995.

[13] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[14] Su-Hui Chiang, Rajesh K. Mansharamani, and Mary K. Vernon. Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 33–44, Nashville, Tennessee, United States, May 1994.

[15] Su-Hui Chiang and Mary K. Vernon. Dynamic vs. static quantum-based parallel processor allocation. In Dror G. Feitelson and Larry Rudolph, editors, *Proceedings of the 2nd Workshop on Job Scheduling Strategies for Parallel Processing*, pages 200–223, Honolulu, Hawaii, United States, April 1996. Springer-Verlag. Lecture Notes in Computer Science Vol. 1162.

[16] Allen B. Downey. Using queue time predictions for processor allocation. In Dror G. Feitelson and Larry Rudolph, editors, *Proceedings of the 3rd Workshop on Job Scheduling Strategies for Parallel Processing*, pages 35–57, Geneva, Switzerland, April 1997. Springer-Verlag. Lecture Notes in Computer Science Vol. 1291.

[17] Dror G. Feitelson. Job scheduling in multiprogrammed parallel systems (extended version). Technical report, IBM Research Report RC 19790 (87657) 2nd Revision, 1997.

[18] Dror G. Feitelson and Larry Rudolph. Parallel job scheduling — a status report. In Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing*, New York, New York, United States, June 2004. To appear in the Springer-Verlag Lecture Notes on Computer Science series.

[19] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998.

[20] R. L. Graham. Bounds for certain multiprocessing anomalies. *The Bell System Technical Journal*, 45:1563–1581, November 1966.

[21] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, March 1969.

[22] Anoop Gupta, Andew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 120–132, San Diego, California, United States, May 1991.

[23] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann, San Francisco, CA, second edition, 1996.

[24] Anna R. Karlin, Kai Li, Mark S. Manasse, and Susan Owicki. Empirical studies of competitive spinning for a shared-memory multiprocessor. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 41–55, Pacific Grove, California, United States, October 1991.

[25] Leonard Kleinrock. Power and deterministic rules of thumb for probabilistic problems in computer communications. In *International Conference on Communications*, volume 3, pages 43.1.1–43.1.10, Boston, Massachusetts, United States, June 1979.

[26] Leonard Kleinrock and Jau-Hsiung Huang. On parallel processing systems: Amdahl's law generalized and some results on optimal design. *IEEE Transactions on Software Engineering*, 18(5):434–447, May 1992.

[27] Scott T. Leutenegger and Mary K. Vernon. The performance of multiprogrammed multiprocessor scheduling policies. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 226–236, Boulder, Colorado, United States, May 1990.

[28] Shau-Ping Lo and Virgil D. Gligor. A comparative analysis of multiprocessor scheduling algorithms. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, Berlin, Germany, September 1987.

[29] Shikharesh Majumdar, Derek L. Eager, and Richard B. Bunt. Scheduling in multiprogrammed parallel systems. In *Proceedings of the 1998 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 104–113, Santa Fe, New Mexico, United States, May 1988.

[30] Shikharesh Majumdar, Derek L. Eager, and Richard B. Bunt. Characterization of programs for scheduling in multiprogrammed parallel systems. *Performance Evaluation*, 13(2):109–130, October 1991.

[31] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. *ACM SIGOPS Operating Systems Review*, 25(5):110–121, 1991.

[32] Xavier Martorell, Julita Corbalán, Dimitrios S. Nikolopoulos, Nacho Navarro, Eleftherios D. Polychronopoulos, Theodore S. Papatheodorou, and Jesús Labarta. A tool to schedule parallel applications on multiprocessors: the NANOS CPU manager. In Dror G. Feitelson and Larry Rudolph, editors, *Proceedings of the 6th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 87–112, Cancun, Mexico, May 2000. Springer-Verlag. Lecture Notes in Computer Science Vol. 1911.

[33] Cathy McCann, Raj Vaswani, and John Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, May 1993.

[34] Robert C. Miller. A type-checking preprocessor for Cilk 2, a multithreaded C language. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1995.

[35] V. K. Naik, M. S. Squillante, and S. K. Setia. Performance analysis of job scheduling policies in parallel supercomputing environments. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 824–833, Portland, Oregon, United States, November 1993.

[36] Thu D. Nguyen, Raj Vaswani, and John Zahorjan. Maximizing speedup through self-tuning of processor allocation. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 463–468, Honolulu, Hawaii, United States, April 1996. IEEE Computer Society.

[37] Thu D. Nguyen, Raj Vaswani, and John Zahorjan. Using runtime measured workload characteristics in parallel processor scheduling. In Dror G. Feitelson and Larry Rudolph, editors, *Proceedings of the 2nd Workshop on Job Scheduling Strategies for Parallel Processing*, pages 155–174, Honolulu, Hawaii, United States, April 1996. Springer-Verlag. Lecture Notes in Computer Science Vol. 1162.

[38] John K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, pages 22–30, Miami/Ft. Lauderdale, Florida, United States, October 1982.

[39] Jitendra D. Padhye and Lawrence Dowdy. Dynamic versus adaptive processor allocation policies for message passing parallel computers: An empirical comparison. In Dror G. Feitelson and Larry Rudolph, editors, *Proceedings of the 2nd Workshop on Job Scheduling Strategies for Parallel Processing*, pages 224–243, Honolulu, Hawaii, United States, April 1996. Springer-Verlag. Lecture Notes in Computer Science Vol. 1162.

[40] Eric W. Parsons and Kenneth C. Sevcik. Multiprocessor scheduling for high-variability service time distributions. In Dror G. Feitelson and Larry Rudolph, editors, *Proceedings of the 1st Workshop on Job Scheduling Strategies for Parallel Processing*, pages 127–145, Santa Barbara, California, United States, April 1995. Springer-Verlag. Lecture Notes in Computer Science Vol. 949.

[41] Eric W. Parsons and Kenneth C. Sevcik. Implementing multiprocessor scheduling disciplines. In Dror G. Feitelson and Larry Rudolph, editors, *Proceedings of the 3rd Workshop on Job Scheduling Strategies for Parallel Processing*, pages 166–192, Geneva, Switzerland, April 1997. Springer-Verlag. Lecture Notes in Computer Science Vol. 1291.

[42] E. Rosti, E. Smirni, L. W. Dowdy, G. Serazzi, and B. M. Carlson. Robust partitioning schemes of multiprocessor systems. *Performance Evaluation*, 19(2-3):141–165, March 1994.

[43] Emilia Rosti, Evgenia Smirni, Giuseppe Serazzi, and Lawrence W. Dowdy. Analysis of non-work-conserving processor partitioning policies. In Dror G. Feitelson and Larry Rudolph, editors, *Proceedings of the 1st Workshop on Job Scheduling Strategies for Parallel Processing*, pages 165–181, Santa Barbara, California, United States, April 1995. Springer-Verlag. Lecture Notes in Computer Science Vol. 949.

[44] Kenneth C. Sevcik. Characterization of parallelism in applications and their use in scheduling. In *Proceedings of the 1989 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 171–180, Berkeley, California, United States, May 1989.

[45] Kenneth C. Sevcik. Application scheduling and processor allocation in multiprogrammed parallel processing systems. *Performance Evaluation*, 19(2-3):107–140, March 1994.

[46] Mark S. Squillante. On the benefits and limitations of dynamic partitioning in parallel computer systems. In Dror G. Feitelson and Larry Rudolph, editors, *Proceedings of the 1st Workshop on Job Scheduling Strategies for Parallel Processing*, pages 219–238, Santa Barbara, California, United States, April 1995. Springer-Verlag. Lecture Notes in Computer Science Vol. 949.

[47] Supercomputing Technologies Group, MIT Laboratory for Computer Science. *Cilk 5.3.2 Reference Manual*, November 2001.

[48] Andrew Tucker and Anoop Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 159–166, Litchfield Park, Arizona, United States, December 1989.

[49] Chansu Yu and Chita R. Das. Limit allocation: An efficient processor management scheme for hypercubes. In *Proceedings of the 1994 International Conference on Parallel Processing*, volume 2, pages 143–150, August 1994.

[50] John Zahorjan, Edward D. Lazowska, and Derek L. Eager. Spinning versus blocking in parallel systems with uncertainty. In *Proceedings of the International Symposium on Performance of Distributed and Parallel Systems*, Kyoto, Japan, December 1988.

[51] John Zahorjan, Edward D. Lazowska, and Derek L. Eager. The effect of scheduling discipline on spin overhead in shared memory parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):180–198, 1991.

[52] John Zahorjan and Cathy McCann. Processor scheduling in shared memory multiprocessors. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 214–225, Boulder, Colorado, United States, May 1990.