Adaptive and Reliable Parallel Computing on Networks of Workstations

Robert D. Blumofe
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
`rdb@cs.utexas.edu`

Philip A. Lisiecki
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, Massachusetts 02139
`lisiecki@mit.edu`

October 21, 1996

Abstract

In this paper, we present the design of *Cilk-NOW*, a runtime system that adaptively and reliably executes functional *Cilk* programs in parallel on a network of UNIX workstations. Cilk (pronounced "silk") is a parallel multithreaded extension of the C language, and all Cilk runtime systems employ a provably efficient thread-scheduling algorithm. Cilk-NOW is such a runtime system, and in addition, Cilk-NOW automatically delivers adaptive and reliable execution for a functional subset of Cilk programs. By adaptive execution, we mean that each Cilk program dynamically utilizes a changing set of otherwise-idle workstations. By reliable execution, we mean that the Cilk-NOW system as a whole and each executing Cilk program are able to tolerate machine and network faults. Cilk-NOW provides these features while programs remain *fault oblivious*, meaning that Cilk programmers need not code for fault tolerance. Throughout this paper, we focus on end-to-end design decisions, and we show how these decisions allow the design to exploit high-level algorithmic properties of the Cilk programming model in order to simplify and streamline the implementation.

1 Introduction

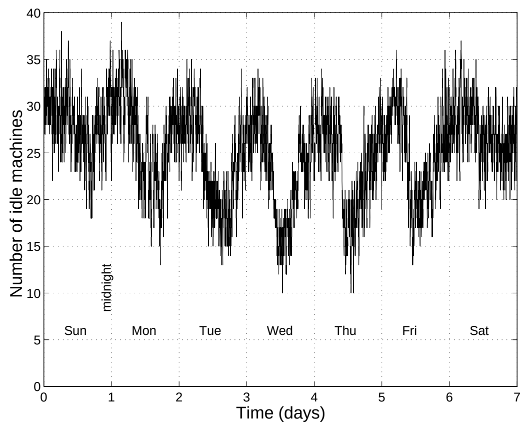
A strong case argues for the use of networks of workstations (NOWs) as parallel-computation platforms [3], and *Cilk-NOW* [6] is a software system that has been designed and implemented to run parallel programs easily and efficiently on networks of UNIX workstations. Implemented entirely in user-level software on top of UNIX, Cilk-NOW is a runtime system for a functional subset of the parallel *Cilk* language [6, 8, 26], a multithreaded extension of C. Applications written in Cilk include graphics rendering, backtrack search, protein folding [37], and the $\star$`Socrates` chess program [25] which won second prize at the 1995 ICCA World Computer Chess Championship running on the 1824-node Intel Paragon at Sandia National Labs. Like all runtime systems for Cilk, Cilk-NOW schedules threads using a provably efficient algorithm based on the technique of random "work stealing" [6, 9] in which processors with no threads steal threads from victims chosen at random. With this algorithm, Cilk delivers performance that is guaranteed to be both efficient and predictable [6, 8]. In addition to thread scheduling, Cilk-NOW also performs *macroscheduling* [30]. That is, Cilk-NOW automatically identifies idle workstations and assigns those idle workstations to help out with running Cilk programs.

The Cilk-NOW runtime system is designed to execute Cilk programs efficiently in the highly dynamic environment of a NOW. Figure 1(a) plots the number of machines that were idle[1] at each point in time over the course of a typical week for a network of 50 SPARCstations at the MIT Laboratory for Computer Science. As can be seen from this plot, though more machines are idle at night, a significant number of machines are idle at various times throughout the day. Therefore, by adaptively using idle machines both day and night, we can take advantage of significantly more machine resources than if we run our parallel jobs as batch jobs during the night. Figure 1(b) is a histogram giving the total idle processor-hours broken down by idle time-interval, from this experiment. This histogram shows that a significant percentage of idle time (1104 processors-hours, or 19.1% of the total 5776 processor-hours) comes from machines that are idle for less than 30 minutes at a time. Thus, the efficient exploitation of idle machines requires that ma-

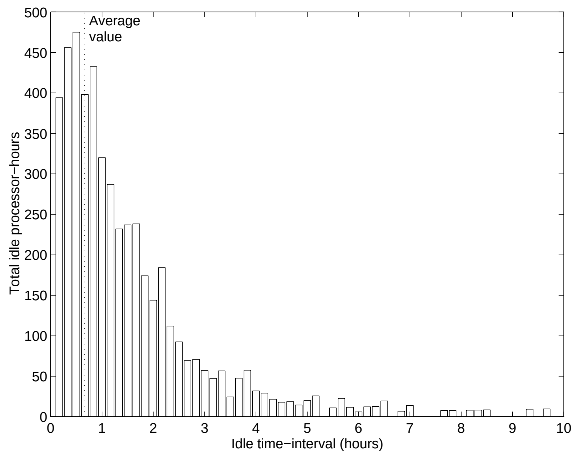
This research was supported in part by the Advanced Research Projects Agency (ARPA) under Grants N00014-94-1-0985 and N00014-92-J-1310. Robert Blumofe was supported in part by an ARPA High-Performance Computing Graduate Fellowship.

This paper appears in the *Proceedings of the USENIX 1997 Annual Technical Symposium*, Anaheim, California, January 6–10, 1997.

[1]For this experiment, a machine is idle if the keyboard and mouse have not been touched for 15 minutes and the 1, 5, and 15 minute processor load averages are below 0.35, 0.30, and 0.25 respectively. These load-average thresholds are reasonable but also somewhat arbitrary.



(a) Idle machines



(b) Idle time

Figure 1: **(a)** This plot shows the number of machines, out of the 50 machines in our network, that were idle at each point in time over the course of one typical week in March, 1995. **(b)** This histogram shows the number of idle processor-hours broken down by idle time-interval. When a machine remains idle for a period of t hours, it contributes t hours to the height of the bar plotted at position t rounded up to the nearest 10 minutes.

chines are able to join and leave a computation quickly and without human intervention. These observations are consistent with those of others [5, 20, 27, 28, 31].

Cilk-NOW provides the following features for running Cilk programs on a network of workstations.

Ease of use. A user can run a Cilk program in parallel on a NOW as if the program were only being run on the local workstation. The user simply types the program's command line, and then the Cilk-NOW runtime system automatically schedules the execution of the program in parallel across the network.

Adaptive parallelism. The Cilk-NOW system adaptively executes Cilk programs on a dynamically changing set of otherwise-idle workstations [6, 10]. When a given workstation is not being used by its owner, the workstation automatically joins in and helps out with the execution of a Cilk program. When the owner returns to work, the machine automatically retreats from the Cilk program.

Fault tolerance. The Cilk-NOW runtime system automatically performs checkpointing, detects failures, and performs recovery [6] while Cilk programs themselves remain *fault oblivious*. That is, Cilk-NOW provides fault tolerance without requiring that programmers code for fault tolerance.

Flexibility. The Cilk-NOW system allows the conditions that are used to determine the idleness of workstations to be set dynamically, in accordance with the tastes of the users and the owners of the machines whose cycles are being stolen. This flexibility preserves the sovereignty of each workstation's owner which is essential to ensure that owners are willing to contribute their workstations for use by others.

Security. The Cilk-NOW system uses secure protocols that do not open a workstation to unauthorized users running foreign code on a machine. The desired degree of security is that which a given system uses to authenticate its remote execution protocol.

Guaranteed performance. The Cilk-NOW system executes Cilk programs using a work-stealing scheduler. This scheduler delivers performance that can be predicted accurately with a simple abstract model [6, 8]. Moreover this simple model can be adapted to the case of heterogeneous processors and networks [32].

Recently, we ran a Cilk protein-folding application `pfold` [37] using Cilk-NOW on a network of about 50 Sun SPARCstations connected by shared 10-Mb/s Ethernet to solve a large-scale protein-folding problem. The program ran for 9 days, surviving several machine crashes and reboots, utilizing 6566 processor-hours of otherwise-idle cycles, with no administrative effort on our part (besides typing `pfold` at the command-line to begin execution), while other users of the network went about their business unaware of the program's presence.

It is important to note that Cilk-NOW provides these features only for Cilk-2 programs which are essentially functional. Cilk-NOW does not support more recent versions of Cilk (Cilk-3 and Cilk-4) that incorporate virtual shared memory, and in particular, Cilk-NOW does not provide any kind of distributed shared memory. In addition, Cilk-NOW does not provide fault tolerance for its I/O facility.

In this paper, we present the design of Cilk-NOW, focusing on those features of Cilk-NOW that are particular to the NOW environment. The Cilk-2 language, work-stealing scheduler, MPP implementation, and guaranteed performance model have been covered at length in other papers [6, 8, 9, 26]. In this paper, we shall focus on adaptive parallelism and fault tolerance. Specifically, we will show how Cilk-NOW's end-to-end design [38] leverages algorithmic properties of the Cilk programming model and work-stealing scheduler in order to amortize all the overhead of adaptive parallelism and fault tolerance against the analytically and empirically bounded overhead of Cilk's work-stealing scheduler.

The remainder of this paper is organized as follows. In Section 2 we review the Cilk-2 language and work-stealing scheduler as first introduced in [8]. In Section 3 we describe the architecture of a Cilk job executing under the Cilk-NOW runtime system. Then, in Section 4 we explain how Cilk-NOW implements adaptive parallelism, and in Section 5 we explain how Cilk-NOW performs checkpointing, fault detection, and fault recovery. In Section 6 we describe the Cilk-NOW macroscheduling system architecture. In Section 7 we compare the Cilk-NOW system to related work. Finally, in Section 8 we outline plans for future work, and we conclude.

2 The Cilk language and work-stealing scheduler

In this section we overview the Cilk parallel multithreaded language and its runtime system's work-stealing scheduler [6, 8, 26]. For brevity, we shall not present the entire Cilk language, and we shall omit some details of the work-stealing algorithm. Since Cilk-2 forms the basis for the Cilk-NOW system, we shall focus on the Cilk-2 language and on the Cilk-2 runtime system as implemented without adaptive parallelism or fault tolerance.

A Cilk program contains one or more *Cilk procedures*, and each Cilk procedure contains one or more *Cilk threads*. A Cilk procedure is the parallel equivalent of a C function, and a Cilk thread is a nonsuspending piece of a procedure. The Cilk runtime system manipulates and schedules the threads. The runtime system is not aware of the grouping of threads into procedures. Cilk procedures are purely an abstraction supported by the `cilk2c` type-checking preprocessor [33].

Consider a program that uses double recursion to compute the Fibonacci function. The Fibonacci function fib(n) for $n \geq 0$ is defined as

$$\text{fib}(n) = \begin{cases} n & \text{if } n < 2; \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{otherwise.} \end{cases}$$

```
thread Fib (cont int k, int n)
{   if (n<2)
        send_argument (k, n);
    else
    {   cont int x, y;
        spawn_next Sum (k, ?x, ?y);
        spawn Fib (x, n-1);
        spawn Fib (y, n-2);
    }
}
```

```
thread Sum (cont int k, int x, int y)
{   send_argument (k, x+y);
}
```

Figure 2: A Cilk procedure to compute the nth Fibonacci number. This procedure contains two threads, `Fib` and `Sum`.

Figure 2 shows how this function is written as a Cilk procedure consisting of two Cilk threads: `Fib` and `Sum`. While double recursion is a terrible way to compute Fibonacci numbers, this toy example does illustrate a common pattern occurring in divide-and-conquer applications: recursive calls solve smaller subcases and then the partial results are merged to produce the final result.

A Cilk thread generates parallelism at runtime by *spawning* a child thread that is the *initial thread* of a child procedure. A spawn is the parallel equivalent of a function call. A spawn differs from a call in that when a thread spawns a child, the parent and child may execute concurrently. After spawning one or more children, the parent thread cannot then wait for its children to return—in Cilk, threads never suspend. Rather, the parent thread must additionally spawn a *successor thread* to wait for the values "returned" from the children. The spawned successor is part of the same procedure as its predecessor. The child procedures return values to the parent procedure by sending those values to the parent's waiting successor. Thus, a thread may wait to begin executing, but once it begins executing, it cannot suspend. This style of interaction among threads is called *continuation-passing style* [4]. Spawning successor and child threads is done with the `spawn_next` and `spawn` keywords respectively. Sending a value to a waiting thread is done with the `send_argument` statement. The Cilk runtime system implements these primitives using two basic data structures: closures and continuations.

Closures are data structures employed by the runtime system to keep track of and schedule the execution of spawned threads. Whenever a thread is spawned, the runtime system allocates a closure for it from a simple heap. A closure consists of a pointer to the code for that thread, a slot for each of the thread's specified arguments, and a *join counter* indicating the number of missing arguments that need to be supplied before the thread is ready to run. The closure, or equivalently the spawned thread, is *ready*

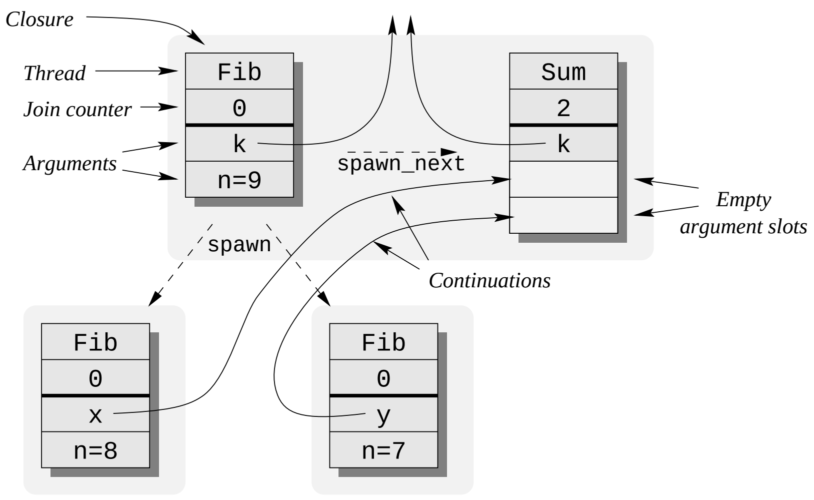


Figure 3: The `Fib` thread spawns a successor and two children. For the successor, it creates a closure with 2 empty argument slots, and for each child, it creates a closure with a continuation referring to one of these empty slots. The background shading denotes Cilk procedures.

if it has obtained all of its arguments, and it is *waiting* if some arguments are missing. To run a ready closure, the Cilk scheduler invokes the thread using the values in the closure as arguments. When the thread dies, the closure is freed.

A *continuation* is a global reference to an empty argument slot of a closure, implemented as a compound data structure containing a pointer to a closure and an offset that designates one of the closure's argument slots. Continuations are typed with the C data type of the slot in the closure. In the Cilk language, continuations are declared by the type modifier keyword `cont`. For example, the `Fib` thread declares two integer continuations, `x` and `y`.

Using the `spawn_next` primitive, a thread spawns a successor thread by creating a closure for the successor. The successor thread is part of the same procedure as its predecessor. For example, in the `Fib` thread, the statement `spawn_next Sum (k, ?x, ?y)` allocates a closure with `Sum` as the thread and three argument slots, as illustrated in Figure 3. The first slot is initialized with the continuation `k` and the last two slots are empty. The continuation variables `x` and `y` are initialized to refer to these two empty slots, and the join counter is set to 2. This closure is waiting.

Similarly, using the `spawn` primitive, a thread spawns a child thread by creating a closure for the child. The child thread is the initial thread of a newly spawned child procedure. The `spawn` statement is semantically identical to `spawn_next`. For example, the `Fib` thread spawns two children as shown in Figure 3. The statement `spawn Fib (x, n-1)` allocates a closure with `Fib` as the thread and two argument slots. The first slot is initialized with the continuation `x` which, as a consequence of the previous statement, refers to a slot in its parent's successor closure. The second slot is initialized with the value of `n-1`. The join counter is set to zero, so the thread is ready.

An executing thread sends a value to a waiting thread by placing the value into an argument slot of the waiting thread's closure. The `send_argument` statement sends a value to the empty argument slot of a waiting closure specified by its argument. The types of the continuation and the value must be compatible. The join counter of the waiting closure is decremented, and if it becomes zero, then the closure is ready. For example, the statement `send_argument (k, n)` in `Fib` writes the value of `n` into an empty argument slot in the parent procedure's waiting `Sum` closure and decrements its join counter. When the `Sum` closure's join counter reaches zero, it is ready. When the `Sum` thread gets executed, it adds its two arguments, `x` and `y`, and then uses `send_argument` to "return" this result up to its parent procedure's waiting `Sum` thread.

At runtime, each processor maintains a "ready" deque (double-ended queue) which contains all of the ready closures. Whenever a closure is created, if its join counter is 0, then it is placed on the head of the ready deque. Whenever a `send_argument` call is made, the join counter is decremented, and if the join counter is decremented to zero, then the closure is placed on the head of the ready deque. When a thread finishes, the next thread to execute is chosen from the head of the ready deque.

If no threads are available in the ready deque, a processor engages in *work stealing*. To steal work, a processor, called the *thief*, chooses another processor, called the *victim*, at random and requests a closure to be sent back. If that processor has any closures in its ready deque, one is removed from the tail of the victim's ready deque and sent across the network to the thief, who will add this closure to its own ready deque. The thief may then begin work on the stolen closure. If the victim has no ready closures, it informs the thief who then tries to steal from another random processor until a ready closure is found or program execution completes.

This simple work-stealing scheduler has been shown, both analytically and empirically, to deliver efficient and predictable performance [6, 8, 9] for "well structured" computations. A *well structured* computation is one in which each procedure sends values (with `send_argument`) only to its parent and only as the last action performed by its last thread. For well structured computations executing on any number P of processors, the execution time can be modeled accurately as $T_1/P + T_\infty$ where T_1 denotes the *work* of the computation—that is, the execution time with 1 processor—and T_∞ denotes the *critical-path length*—that is, the theoretical execution time on an ideal machine with infinitely many processors. Such performance is within a factor of 2 of optimal, and additionally when the critical path is short compared to the amount of work per processor, such performance displays *linear speedup*.

The key element in proving this $T_1/P + T_\infty$ performance bound is the fact that closures are always stolen from the tail of the ready deque. For well structured computations, a closure that is on the critical path must be at the tail of some processor's ready deque. Thus, when processors are not executing closures, they are stealing work and, therefore, are likely to be making progress on the critical path. As a corollary to this result, the number of work-steal attempts per processor is proportional to the critical-path length and does not grow with the work. Thus, a computation with a sufficiently short critical path compared to the work per processor can continue to display linear speedup even when communication is very expensive. This idea of amortizing overhead against the critical path plays an important role in our later discussion of adaptive parallelism and fault tolerance.

3 Cilk-NOW job architecture

The Cilk-NOW runtime system consists of several component programs that (in addition to macroscheduling duties discussed later) manage the execution of each individual Cilk program. In this section, we shall cover the architecture of a Cilk program as it is executed by the Cilk-NOW runtime system, explaining the operation of each component and their interactions.

In Cilk-NOW terminology, we refer to an executing Cilk program as a Cilk *job*. Since Cilk programs are parallel programs, a Cilk job consists of several processes running on several machines. One process, called the *clearinghouse*, in each Cilk job runs a system-supplied program called `CilkChouse` that is responsible for keeping track of all the other processes that comprise a given job. These other processes are called *workers*. A worker is a process running the actual executable of a Cilk program. Since Cilk jobs are adaptively parallel, the set of workers is dynamic. At any given time during the execution of a job, a new worker may join the job or an existing worker may leave. Thus, each Cilk job consists of one or more workers and a clearinghouse to keep track of them.

The Cilk-NOW runtime system contains additional components that perform macroscheduling as discussed in Section 6, but for the purpose of our present discussion, we need only introduce the "node managers." A *node manager* is a process running a system-supplied program called `CilkNodeManager`. A node manager runs as a background daemon on every machine in the network. It continually monitors its machine to determine when the machine is idle.

To see how all of these components work together in managing the execution of a Cilk job, we shall run through an example. (In describing interactions with the macroscheduler, we shall refer to the macroscheduler as a single entity, though actually, as we shall see in Section 6, the macroscheduler is a distributed subsystem with several components.) Suppose that a user sits down at a machine called `Penguin` to run the `pfold` program. In our example, the user types

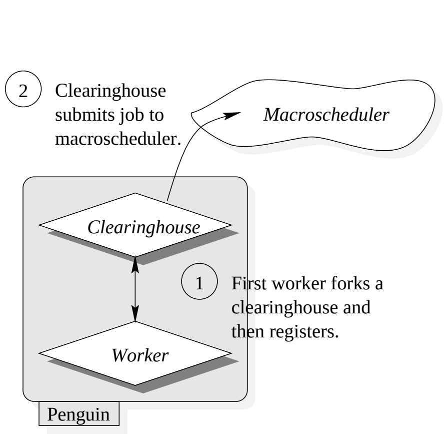
```
pfold 3 7
```

at the shell, thereby launching a Cilk job to enumerate all protein foldings using 3 initial folding sequences and starting with the 7th one.

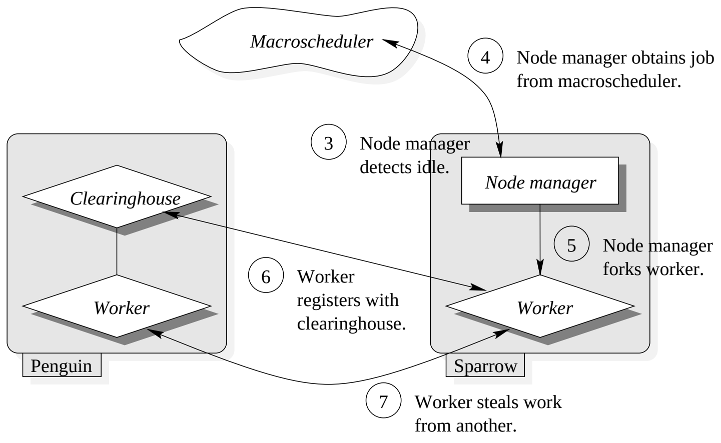
The new Cilk job begins execution as illustrated in Figure 4(a). The new process running the `pfold` executable is the first worker and begins execution by forking a clearinghouse with the command line

```
CilkChouse -- pfold 3 7.
```

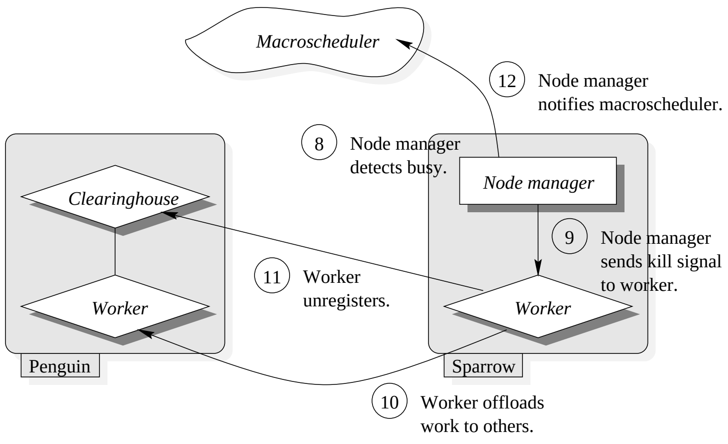
Thus, the clearinghouse knows that it is in charge of a job whose workers are running "`pfold 3 7`." The clearinghouse begins execution by sending a *job description* to the macroscheduler. The job description is a record containing several fields. Among these fields is the name of the Cilk program executable—in this case `pfold`—and the clearinghouse's network address. The clearinghouse then goes into a service loop waiting for messages from its workers. After forking the clearinghouse, the first worker *registers* with the clearinghouse by sending it a message containing its own network address. Now the clearinghouse knows about one worker, and it responds to that worker by assigning it a unique *name*. Workers are named with numbers, starting with number `0`. Having registered, worker `0` begins executing the Cilk program as described in Section 2. We now have a running Cilk job with one worker.


2 Clearinghouse submits job to macroscheduler.
Macroscheduler
Clearinghouse
1 First worker forks a clearinghouse and then registers.
Worker
Penguin

(a) A Cilk job starts.


Macroscheduler
4 Node manager obtains job from macroscheduler.
3 Node manager detects idle.
Node manager
Clearinghouse
5 Node manager forks worker.
6 Worker registers with clearinghouse.
Worker
Worker
Penguin
Sparrow
7 Worker steals work from another.

(b) An idle machine joins the job.


Macroscheduler
12 Node manager notifies macroscheduler.
8 Node manager detects busy.
Node manager
Clearinghouse
9 Node manager sends kill signal to worker.
11 Worker unregisters.
Worker
Worker
Penguin
Sparrow
10 Worker offloads work to others.

(c) A no-longer-idle machine leaves the job.

Figure 4: **(a)** The first worker forks a clearinghouse, and then the clearinghouse submits the job to the macroscheduler. **(b)** When the node manager detects that its machine is idle, it obtains a job from the macroscheduler and then forks a worker. The worker registers with the clearinghouse and then begins work stealing. **(c)** When the node manager detects that its machine is no-longer idle, it sends a kill signal to the worker. The worker catches this signal, offloads its work to other workers, unregisters with the clearinghouse, and then terminates.

A second worker joins the Cilk job when some other workstation in the network discovers that it is idle, as illustrated in Figure 4(b). Suppose the node manager on a machine named `Sparrow` detects that the machine is idle. The node manager sends a message to the macroscheduler, and the macroscheduler responds with the job description of a Cilk job for the machine to work on. In this case, the job description specifies our `pfold` job by giving the name of the executable—`pfold`—and the network address of the clearinghouse. The node manager then uses this information to fork a new worker as a child with the command line

```
pfold -NoChouse
-Address=clearinghouse-address
--.
```

The `-NoChouse` flag on the command line tells the worker that it is to be an additional worker in an already existing Cilk job. (Without this flag, the worker would fork a new clearinghouse and start a new Cilk job.) The `-Address` field on the command line tells the worker where in the network to find the clearinghouse. The worker uses this address to send a registration message, containing its own network address, to the clearinghouse. The clearinghouse responds with the worker's assigned name—in this case, number `1`—and the job's command-line arguments—in this case, "`pfold 3 7`." Additionally, the clearinghouse responds with a list of the network addresses of all other registered workers. Now the new worker knows the addresses of the other workers, so it can commence execution of the Cilk program and steal work as described in Section 2. We now have a running Cilk job with two workers.

Now, suppose that someone touches the keyboard on `Sparrow`. In this case, the node manager detects that the machine is busy, and the machine leaves the Cilk job as illustrated in Figure 4(c). After detecting that the machine is busy, the node manager sends a kill signal to its child worker. The worker catches this signal and prepares to leave the job. First, the worker offloads all of its closures to other workers as explained in more detail in Section 4. Next, the worker sends a message to the clearinghouse to *unregister*. Finally, the worker terminates.

When a Cilk job is running, each worker periodically checks in with the clearinghouse. Specifically, each worker periodically (every 2 seconds) sends a message to the clearinghouse, and the clearinghouse responds with an *update* message informing the worker of any other workers that have left the job and any new workers that have joined the job. For each new worker that has joined, the clearinghouse also provides the network address. If the clearinghouse does not receive any messages from a given worker for an extended period of time (30 seconds), then the clearinghouse determines that the worker has crashed. In later update messages, the clearinghouse informs the other workers of the crash, and the other workers take appropriate remedial action as described in Section 5.

All communication between workers, and between workers and the clearinghouse, is implemented with UDP/IP [13, 40]. Knowing that UDP datagrams are unreliable, the Cilk-NOW protocols incorporate appropriate mechanisms, such as acknowledgments, retries, and timeouts, to ensure correct operation when messages get lost. We shall not discuss these mechanisms in any detail, and in order to simplify our exposition of Cilk-NOW, we shall often speak of messages being sent and received as if they are reliable. What we will say about these mechanisms is that they are built on top of UDP but without any effort to create a reliable message-passing layer. Rather these mechanisms are built directly into the runtime system's protocols, so in the common case when a message does get through, Cilk-NOW pays no overhead to make the message reliable.

We chose to build Cilk-NOW's communication protocols using an unreliable message-passing layer instead of a reliable one for two reasons, both based on end-to-end design arguments [38]. First, reliable layers such as TCP/IP [40] and PVM [41] perform implicit acknowledgments and retries to achieve reliability. Therefore, such layers either preclude the use of asynchronous communication or require extra buffering and copying. A layer such as UDP which provides minimal service guarantees can be implemented with considerably less software overhead than a layer with more service features. In the common case when the additional service is not needed, the minimal layer can easily outperform its fully-featured counterpart. Second, in an environment where machines can crash and networks can break, the notion of a "reliable" message-passing layer is somewhat suspect. A runtime system operating in an inherently unreliable environment cannot expect the message-passing layer to make the environment reliable. Rather, the runtime system must incorporate appropriate mechanisms into its protocols to take action when a communication endpoint or link fails. For these reasons, we chose to build the Cilk-NOW runtime system on top of a minimal layer of message-passing service and incorporate mechanisms directly into the runtime system's protocols in order to handle issues of reliability. The downside to this approach is complexity. The protocols implemented in the Cilk-NOW runtime system are complex: the code for these protocols takes almost 20 percent of the total runtime-system code, and the programming effort was probably near half of the total. Nevertheless, this was a one-time effort that we expect will reap performance rewards for a long time to come.

4 Adaptive parallelism

Adaptive parallelism allows a Cilk job to take advantage of idle machines whether or not they are idle when the job starts and whether or not they will remain idle for the duration of the job. In order to efficiently utilize machines that may join and leave a running job, the overhead of supporting this feature must not excessively slow down the work of any worker at a time when it is not joining or leaving. As we saw in the previous section, a new worker joins a job easily enough by registering with the clearinghouse and then stealing a closure. A worker leaves a job by migrating all of its closures to other workers, and here the danger lies. When we migrate a waiting closure, other closures with continuations that refer to this closure must somehow update these continuations so they can find the waiting closure at its new location. (Without adaptive parallelism, waiting closures never move.) Naively, each migrated waiting closure would have to inform every other closure of its new location. In this section, we show how we can take advantage of Cilk's well structuring and the work-stealing scheduler to make this migration extremely simple and efficient. (Experimental results documenting the efficiency of Cilk-NOW's adaptive parallelism have been omitted for lack of space but can be found in [6].)

Our approach is to impose additional structure on the organization of closures and continuations, such that the structure is cheap to maintain while simplifying the migration of closures. Specifically, we maintain closures in "subcomputations" that migrate en masse, and every continuation in a closure refers to a closure in the same subcomputation. In order to send a value from a closure in one subcomputation to a closure in another, we forward the value through intermediate "result closures," and give each result closure the ability to send the value to precisely one other closure in one other subcomputation. With this structure and these mechanisms, all of the overhead associated with adaptive parallelism (other than the actual migration of closures) occurs only when closures are stolen, and as we saw in Section 2, the number of steals grows at most linearly with the critical path of the computation and is not a function of the work. The bulk of this section's exposition concerns the organization of closures in subcomputations and the implementation of continuations. After covering these topics, the mechanism by which closures are migrated to facilitate adaptive parallelism is quite straightforward.

In Cilk-NOW, every closure is maintained in one of three pools associated with a data structure called a *subcomputation*. A subcomputation is a record containing (among other things) three pools of closures. The *ready pool* is the list of ready closures described in Section 2. The *waiting pool* is a list of waiting closures. The *assigned pool* is a list of ready closures that have been stolen away. Program execution begins with one subcomputation—the *root* subcomputation—allocated by worker `0` and containing a single closure—the initial thread of `cilk_main`—in the ready pool. In general, a subcomputation with any closures in its ready pool is said to be *ready*, and ready subcomputations can be executed by the scheduler as described in Section 2 with the additional provision that each waiting closure is kept in the waiting pool and then moved to the ready pool when its join counter decrements to zero. The assigned pool is used in work stealing as we shall now see.

The act of work stealing creates a new subcomputation on the thief which is linked to a copy of the stolen closure kept in an assigned pool on the victim. If a worker needs to steal work, then before sending a steal request to a victim, it allocates a new subcomputation from a simple runtime heap and gives the subcomputation a unique *name*. The subcomputation's name is formed by concatenating the thief worker's name and a number unique to that worker. The first subcomputation allocated by a worker `r` is named `r:1`, the second is named `r:2`, and so on. The root subcomputation is named `0:1`. The steal request message contains the name of the thief's newly allocated subcomputation. When the victim worker gets the request message, if it has any ready subcomputations, then it chooses a ready subcomputation in round-robin fashion, removes the closure at the tail of the subcomputation's ready pool, and places this *victim closure* in the assigned pool. As illustrated in Figure 5, the victim worker then *assigns* the closure to the thief's subcomputation by adding to the closure an *assignment information* record allocated from a simple runtime heap, and then storing the name of the thief worker and the name of the thief's subcomputation (as contained in the steal request message) in the assignment information. Finally, the victim worker sends a copy of the closure to the thief. When the thief receives the stolen closure, it records the name of the victim worker in its subcomputation, and it places the closure in the subcomputation's ready pool. Now the thief's subcomputation is ready, and the thief worker may commence executing it. Notice that the victim closure and thief subcomputation can refer to each other via the thief subcomputation's name which is stored both in the victim closure's assignment information and in the thief subcomputation, as illustrated in Figure 5.

When a worker finishes executing a subcomputation, the link between the subcomputation and its victim closure is destroyed. Specifically, when a subcomputation has no closures in any of its three pools, then the subcomputation is *finished*. A worker with a finished subcomputation sends a message containing the subcomputation's name to the subcomputation's victim worker. Using this name, the victim worker finds the victim closure. This closure is removed from its subcomputation's assigned pool and then the closure and its assignment information are freed. The victim worker then acknowledges the message, and when the thief worker receives the acknowledgment, it frees its subcomputation. When the root subcomputation is finished, the entire Cilk job is finished.

In addition to allocating a new subcomputation, whenever a worker steals a closure, it also allocates a new "result" closure, and it alters the continuation in the stolen closure so that it refers to the result closure. Consider a thief stealing a closure, and suppose the victim closure contains a continuation referring to a closure that we call the *target*. (The victim and target closures must be in the same subcomputation in the victim worker.) Continuations are implemented as the address of the target closure concatenated with the index of an argument slot in the target closure. Therefore, the continuation in the victim closure contains the address of the target closure, and this address is only meaningful to the victim worker. Thus, when the thief worker receives the stolen closure, it replaces the continuation with a new continuation referring to an empty slot in a newly allocated *result* closure. The stolen and result closures are part of the same subcomputation. The result closure's thread is a special system thread whose operation we shall explain shortly. This thread takes one argument: a *result value*. The result value is initially missing, and the continuation in the stolen closure is set to refer to this argument slot. The result closure is waiting and its join counter is 1.

Using continuations to send values from one thread to another operates as described in Section 2, but when a value is sent to a result closure, communication between

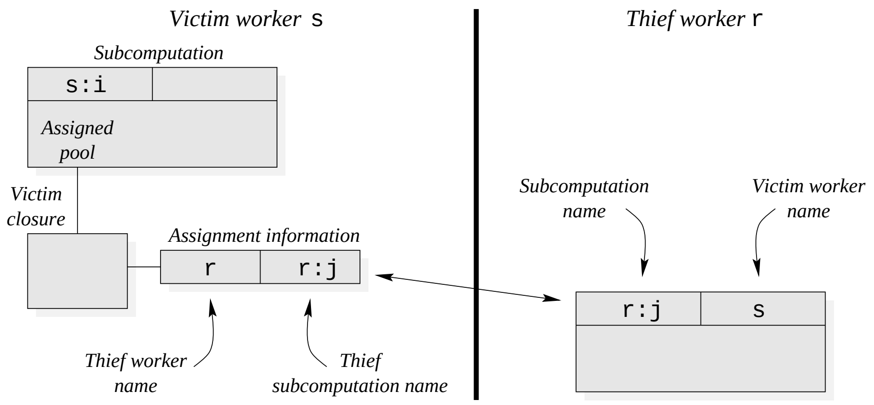


Figure 5: A victim closure stolen from the subcomputation `s:i` of victim worker `s` is assigned to the thief subcomputation `r:j`. The victim closure is placed in the assigned pool and augmented with assignment information that records the name of the thief worker and the name of the thief subcomputation. The thief subcomputation records its own name and the name of the victim worker. Thus, the victim closure and thief subcomputation can refer to each other via the thief subcomputation's name.

different subcomputations occurs. When a result closure receives its result value, it becomes ready, and when its thread executes, it forwards the result value to another closure in another subcomputation as follows. When a worker executing a subcomputation executes a result closure's thread, it sends a message to the subcomputation's victim worker. This message contains the subcomputation's name as well as the result value that is the thread's argument. When the victim worker receives this message, it uses the subcomputation name to find the victim closure, and then it uses the continuation in the victim closure to send the result value to the target.

To summarize, each subcomputation contains a collection of closures and every continuation in a closure refers to another closure in the same subcomputation. To send a value from a closure in one subcomputation to a closure in another, the value must be forwarded through an intermediate result closure.

With this structure, migrating a subcomputation from one worker to another is fairly straightforward. At the source worker, the entire subcomputation is "pickled" by giving each of the subcomputation's closures a number and replacing each continuation's pointer with the corresponding number. Then, after sending the closures to the destination worker, the destination worker reconstructs the subcomputation by reversing the pickling operation. The subcomputation keeps its name, so after a migration, the first component of the subcomputation name will be different than the name of the worker. When the subcomputation and all of its closures have been migrated to their destination worker, this worker sends a message to the subcomputation's victim worker to inform the victim closure of its thief subcomputation's new thief worker. Additionally, for each of the subcomputation's assigned closures, it sends a message to the thief worker to inform the thief subcomputation of its victim closure's new victim worker. Thus, all of the links between victim closures and thief subcomputations are restored.

5 Fault tolerance

With transparent fault tolerance built into the Cilk-NOW runtime system, Cilk jobs may survive machine crashes or network outages despite the fact that Cilk programs are fault oblivious, having been coded with no special provision for handling machine or network failures. If a worker crashes, then other workers automatically redo any work that was lost in the crash. In the case of a more catastrophic failure, such as a power outage, a total network failure, or a crash of the file server, then all workers may crash. For this case, Cilk-NOW provides automatic checkpointing, so when service is restored, the Cilk job may be restarted with minimal lost work. Recall that Cilk-NOW does not provide fault tolerance for I/O.

In this section, we show how the structure used to support adaptive parallelism—which leverages Cilk's tree structure and the work-stealing scheduler—may be further leveraged to build these fault tolerant capabilities in Cilk-NOW. As with adaptive parallelism, all of the overhead associated with fault tolerance (other than the cost of periodic checkpoints) can be amortized against the number of steals which grows at most linearly with the critical path and is not a function of the work.

Given adaptive parallelism, fault tolerance is only a short step away. With adaptive parallelism, a worker may leave a Cilk job, but before doing so, it first migrates all of its subcomputations to other workers. In contrast, when a worker crashes, all of its subcomputations are lost. To support fault tolerance, we add a mechanism that allows surviving workers to redo any work that was done by the lost subcomputations. Such a mechanism must address two fundamental issues. First, not all work is necessarily idempotent, so redoing work may present problems. We address this issue with a technique that we call a *return transaction*. Specifically, we ensure that the work done by any given subcomputation does not affect the state of any other subcomputations until the given subcomputation finishes. Thus, from the point-of-view of any other subcomputation, the work of a subcomputation appears as a transaction: either the subcomputation finishes and commits its work by making it visible to other subcomputations, or the subcomputation never happened. Second, the lost subcomputations may have done a large amount of work, and we would like to minimize the amount of work that needs to be redone. We address this issue by incorporating a transparent and fully distributed checkpointing facility. This checkpointing facility also allows a Cilk job to be restarted in the case of a total system failure in which every worker crashes.

To turn the work of a subcomputation into a return transaction, we modify the behavior of the subcomputation's result closure. In Cilk, returning a value is always the last operation performed by a Cilk procedure, so the result closure cannot be ready until the subcomputation is finished. In addition, recall that the execution of the result closure and the finishing of the subcomputation both warrant a message to the victim worker. Thus, we bundle these two messages into a single larger message sent to the victim worker. When the victim worker receives this message, it commits all of the thief subcomputation's work by sending the appropriate result value from the victim closure, freeing the victim closure (and its assignment information), and sending an acknowledgment back to the thief worker.

With subcomputations having this transactional nature, a Cilk job can tolerate individual worker crashes as follows. Suppose a worker crashes. Eventually, the clearinghouse will detect the crash, and the other living workers will learn of the crash at the next update from the clearinghouse. When a worker learns of a crash, it goes through all of its subcomputations, checking each assigned closure to see if it is assigned to the crashed worker. Each such closure is moved from the assigned pool back to the ready pool (and its assignment information is freed). Thus, all of the work done by the closure's thief subcomputation which has been lost in the crash will eventually be redone. Additionally, when a worker learns of a crash, it goes through all of its subcomputations to see if it has any that record the crashed worker as the subcomputation's victim. For each such subcomputation, the worker aborts it as follows. The worker goes through all of the subcomputation's assigned closures sending to each thief worker an *abort* message specifying the name of the thief subcomputation. Then the worker frees the subcomputation and all of its closures. When a worker receives an abort message, it finds the thief subcomputation named in the message and recursively aborts it. All of the work done by these aborted subcomputations must eventually be redone. In order to avoid aborting all of these subcomputations (which may comprise the entire job in the case when the root subcomputation is lost) and redoing potentially vast amounts of work, and in order to allow restarting when the entire job is lost, we need checkpointing.

Cilk-NOW performs automatic checkpointing without any synchronization among different workers and without any notion of global state. Specifically, each subcomputation is periodically checkpointed to a file named with the subcomputation's name. For example, a subcomputation named `r:i` would be checkpointed to a file named `scomp_r_i`. We assume that all workers in the job have access to a common file system (through NFS or AFS, for example), and all checkpoint files are written to a common checkpoint directory.[2] To write a checkpoint file for a subcomputation `r:i`, the worker first opens a file named `scomp_r_i.temp`. Then, it writes the subcomputation record and all of the closures—including the assignment information for the assigned closures—into the file. Finally, it atomically renames the file `scomp_r_i.temp` to `scomp_r_i`, overwriting any previous checkpoint file. A checkpoint file can be read to recover the subcomputation. On writing a checkpoint file, the worker additionally prunes any no-longer-needed checkpoint files.

If workers crash, the lost subcomputations can be recovered from checkpoint files. In the case of a single worker crash, the lost subcomputations can be recovered automatically. When a surviving worker finds that it has a subcomputation with a closure assigned to the crashed worker, then it can recover the thief subcomputation by reading the checkpoint file. In the case of a large-scale failure in which every worker crashes, the Cilk job can be restarted from checkpoint files by setting the `-Recover` flag on the command line. Recovery begins with the root subcomputation whose checkpoint file is `scomp_0_1`. After recovering the root subcomputation, then every other subcomputation can be recovered by recursively recovering the thief subcomputation for each of the root subcomputation's assigned closures.

6 Cilk-NOW macroscheduling

The Cilk-NOW runtime system contains components that perform macroscheduling [30]. The macroscheduler identifies idle machines and determines which machines work on which jobs. In this section, we discuss each component of the macroscheduler, and we show how

[2]We have not yet implemented any sort of distributed file system. In the current implementation, workers implicitly synchronize when they write checkpoint files, since they all access a common file system.

they work together.

Like the workers and the clearinghouse, which together comprise a single parallel Cilk job, the components of the macroscheduler are distributed across the network. As already mentioned, each machine in the network runs a node manager, an instance of the `CilkNodeManager` program, that monitors the machine's idleness and the status of a worker if one is present. In addition to the clearinghouse, each Cilk job executes a single *job manager*, an instance of the `CilkJobManager` program, that services requests for the job description. Each of these components registers with a central *job broker*, an instance of the `CilkJobBroker` program. The job broker keeps track of the set of node managers and job managers running in the network. The Cilk-NOW runtime system can continue operation even if some of these components, including the job broker, fail.

Each machine in the network runs a node manager that is responsible for determining when the machine is idle. When the machine is being used, the node manager wakes up every 5 seconds to determine if the machine has gone idle. It looks at how much time has elapsed since the keyboard and mouse have been touched, the number of users logged in, and the processor load averages. The node manager then passes these values through a predicate to decide if the machine is idle. A typical predicate might require that the keyboard and mouse have not been touched for at least 5 minutes and the 1-minute processor load average is below 0.35. The predicate can be customized for each machine. We believe that maintaining the owner's sovereignty is essential if we want owners to allow their machines to be used for parallel computation. A user can change a predicate with a simple command-line utility called `CilkPred`. For example, issuing the command

```
CilkPred user=lisiecki global add
idletime=900
```

causes any workstation on the network to require that the user "lisiecki" be idle for at least 900 seconds. Alternatively, a user might issue the command

```
CilkPred always node=vulture add
load=.2,.2,.2
```

which applies only to the workstation `Vulture` and requires it to have a load average of 0.2 or less for all of the 1, 5, and 15 minute load averages.

When all applicable conditions of the predicate are satisfied, the machine is idle, and the node manager obtains a job description from a job manager (using an address given by the job broker or another node manager) and forks a worker. The node manager then monitors the worker and continues to monitor the machine's idleness. With a worker running, the node manager wakes up once every second to determine if the machine is still idle (adding an estimate of the running job's processor usage to any processor load-average threshold). If the machine is no longer idle, then the node manager sends a kill signal to the worker as previously described. When the worker process dies for any reason, the node manager takes one of two possible actions. If the machine is still idle, then it obtains a new job description and forks a new worker. If the machine is no longer idle, then it returns to monitoring the machine once every 5 seconds.

When a Cilk job begins execution, a job manager is started automatically by the clearinghouse. The clearinghouse submits the job description to the job manager as alluded to in Section 3, and then the job manager registers itself with the job broker. The job manager then goes to sleep, and it periodically wakes up to reregister with the job broker in case the job broker has crashed and restarted. When the job terminates, the job manager unregisters with the job broker. The job manager is the central authorizing agent for the job. Any time a node manager forks a worker, it receives a copy of the job description directly from the job's job manager.

When a node manager forks a new worker, it must take special precautions that the user specified in the job description actually authorized the job to be run. Failure to do so would allow an outsider to gain unauthorized access to a user's account. Furthermore, it is desirable for the macroscheduler's protocols to be secure against unauthorized messages. For these reasons, all of the macroscheduler's protocols are secured with an abstraction on top of UDP called *secure active messages* [30]. This abstraction maintains all of the semantics of the split-phase protocols mentioned earlier but adds a guarantee of the authenticity of messages to the receiver. Unlike a normal UDP message which is sent from one network address to another, a secure active message is sent between "principals." A *principal* is a pair consisting of a network address and a claim as to the identity of the sender. Each secure active message contains user data, the sending principal, and whatever additional data might be required by the underlying authentication protocol, whether that be the standard UNIX `rsh` protocol or a protocol like Kerberos [34]. The security layer is very simplistic, providing only enough functionality to allow the protocols to be secured in a manner independent of the authentication protocol.

The decision to receive the job description directly from the job manager stems from security considerations with protocols like Kerberos, where the job broker and node managers are not trusted with the user's credentials. Only the user in the possession of tickets can be trusted to start a remote process. Since the job manager runs as the desired user, retrieving the job description directly and securely from the job manager assures that the user has actually authorized running the job.

The task of scheduling jobs on workstations is shared between the job broker and the node managers. The job broker is responsible for ensuring that each job is running on at least one workstation. The node managers then use a distributed, randomized algorithm to divide the workstations evenly among the jobs. Because the node managers are capable of performing scheduling, temporary outages of the job broker do not impede progress in scheduling jobs on the network of workstations. We are currently experimenting with a distributed, randomized macroscheduling algorithm that uses steal rates to estimate worker utilization. Each job should get its fair share of the idle machines, but no job should get more machines than it can efficiently utilize.

7 Related work

Cilk-NOW is unique in delivering adaptive and reliable execution for parallel programs on networks of workstations. Traditionally, systems such as PVM [41], TreadMarks [2], and others [11, 16, 23, 29] that are designed to support parallel programs on networks of workstations have not provided adaptive parallelism or fault tolerance. On the other hand, most systems that do provide support for adaptive execution or fault tolerance take a "process-centric" approach. That is, they provide an abstraction of mobile processes and/or an abstraction of reliable processes. As such these systems are very general in their potential application, but they do not provide much support for parallel programs. In contrast, Cilk-NOW does provide support for parallel programs and it does provide adaptive parallelism and fault tolerance, but it does so only for the Cilk parallel programming model. Such specificity allows the Cilk-NOW design to take an end-to-end approach [38] that leverages properties of the Cilk programming model in order to implement adaptive parallelism and fault tolerance simply and efficiently.

Distributed operating systems [17, 36, 43, 46] and remote execution facilities [18, 19, 31, 35, 47] provide services such as remote process execution and, in some cases, process migration. These systems are not intended to be parallel programming environments, though presumably a parallel programming environment could by built atop one of these systems. In fact, Orca [42], which has been built on top of Amoeba, is such a system. These systems are process-centric in that they adapt only by remotely executing and/or migrating processes.

A small number of parallel programming and runtime systems have been built that are adaptively parallel, but unlike Cilk-NOW, none are fault tolerant. Possibly the first adaptively parallel system is the Benevolent Bandit Laboratory (BBL) [22], and Cilk-NOW borrows some of its overall system architecture from BBL. The Piranha system [24, 27], which is based on the Linda programming model [12], is also adaptively parallel. (In fact, the authors of Piranha appear to have coined the term "adaptively parallel.") These systems support programming models that are quite different from Cilk's, but as with the Cilk-NOW design, both leverage properties of their programming model in order to implement adaptive parallelism. A runtime system for the programming language COOL [14] running on symmetric multiprocessors [44] and cache-coherent, distributed, shared-memory machines uses process control to support adaptive parallelism. This system relies on special-purpose operating system and hardware support. In contrast, Cilk-NOW supports adaptive parallelism entirely in user-level software on top of commercial hardware and operating systems. The Spawn system [45] supports concurrent applications with dynamic and adaptive resource management policies based on microeconomic principles. Unlike Cilk-NOW, none of these systems are fault tolerant.

A growing number of systems do provide fault tolerance, but unlike Cilk-NOW, none provide "application" fault tolerance in a high-level parallel programming environment. The Hive [15] distributed operating system provides "system" fault tolerance, meaning that a fault in one component does not bring down the entire system. Hive does not, however, provide "application" fault tolerance, meaning that with Hive, if an application is using a failed component, then the entire application crashes (unless the application itself has taken care to be fault tolerant). Application fault tolerance is provided by the Manetho system [21] via the technique of message logging. The Sam system [39] uses message logging to implement a fault tolerant distributed shared memory. The Sam implementation leverages properties of its shared-memory consistency model in order to avoid logging certain messages. Unlike Cilk-NOW, both Manetho and Sam are process-centric, as they both provide the abstraction of reliable processes, and neither is really a high-level parallel programming environment.

In comparing Cilk-NOW with these other process-centric systems, an interesting question to ask is, why not build the Cilk-NOW runtime system on top of one these other systems? After all, these systems already implement adaptive and/or fault-tolerant execution. The answer is performance. As we have seen, the overhead of adaptive parallelism and fault tolerance in Cilk-NOW is amortized against the overhead in Cilk's provably efficient scheduling algorithm. This amortization is only possible because all facets of the design are specialized with high-level knowledge of algorithmic structure in the Cilk programming model.

8 Conclusion

The widespread use of NOWs for parallel computation requires a software infrastructure that allows programmers to code in a high-level language that abstracts away the complexity of protocols, scheduling, and resource management. Cilk and Cilk-NOW are part of this developing software infrastructure. In this paper, we have shown how Cilk-NOW's end-to-end design leverages structure in the Cilk programming model to implement adaptive parallelism and fault tolerance simply and efficiently. All overheads are amortized against work-stealing operations, and the number of steals grows with the critical path and not with the work. This result is only possible because Cilk-NOW incorporates Cilk-specific policies at all levels of its design.

The Cilk-NOW runtime system, as described in this paper and as currently implemented, supports the Cilk-2 language which is essentially functional in that it does not have support for a global address space or parallel I/O. More recent incarnations of Cilk for MPPs and SMPs have support for a global address space using "dag-consistent" distributed shared memory [7], and we are currently working on extensions for parallel I/O. With these additions to Cilk, preserving Cilk-NOW's adaptive and fault tolerant execution model remains a challenging open problem. We are currently working on this problem. The dag-consistency model was conceived with adaptive parallelism and fault tolerance in mind, and we are investigating the idea of coupling our current return transactions mechanism with a causal message-logging mechanism [1].

In other current research, we are investigating distributed macroscheduling algorithms. The goal of such an algorithm is to assign idle workstations to Cilk jobs so that each job gets a "fair" share and without requiring that users explicitly state their application's resource needs. It turns out that the parallelism of a Cilk job can be determined continuously and automatically by monitoring the steal rate. We are examining a macroscheduling algorithm in which this information is used in randomized pairwise interactions among processors. The idea is that periodically (and asynchronously) each processor picks another processor in the network at random, and if the two processors are working on different jobs, then one processor may switch to the job of the other. The switching decision is randomized and based only on information about the parallelism and size of the two jobs involved. Early simulation results indicate that such a scheme is very effective [30], and we are currently working on analysis and implementation.

More information about Cilk, including papers, documentation, and software releases, but not including Cilk-NOW software, can be found on the World-Wide Web at `http://theory.lcs.mit.edu/~cilk`.

Acknowledgments

We wish to thank the entire Cilk project team at MIT. Led by Professor Charles E. Leiserson, this team includes or has included Matteo Frigo, Michael Halbherr now of the Boston Consulting Group, Chris Joerg now of DEC's Cambridge Research Lab, Bradley Kuszmaul now of Yale University, Howard Lu, Rob Miller now of Carnegie Mellon University, David Park now of McKinsey and Associates, Keith Randall, and Yuli Zhou now of AT&T Labs Research. Of particular note in this group, we wish to extend an extra note of gratitude to David Park, who helped conceive and write the very first Cilk-NOW prototype (back then we called the system "Phish") and especially to Charles Leiserson. From its conception, Charles has supported this project with a generous allocation of resources, both human and machine, and with his enthusiasm, ideas, and sense of humor.

Other current and former members of MIT's Laboratory for Computer Science who we wish to thank include Arvind, Scott Blomquist, Eric Brewer now of the University of California at Berkeley, Frans Kaashoek, Larry Rudolph, and Sivan Toledo now of IBM.

Finally, we wish to thank the anonymous reviewers for their many helpful comments and our "shepherd," Fred Douglis, for his patience and guidance in helping us address the reviewers' comments.

References

[1] Lorenzo Alvisi and Keith Marzullo. Message logging: Pessimistic, optimistic, causal and optimal. In *Proceedings of the 15th IEEE International Conference on Distributed Computing Systems*, pages 229–236, Vancouver, Canada, June 1995.

[2] Cristiana Amza, Alan L. Cox, Sandhya Dwarkadas, Pete Keleher, Honghui Lu, Ramakrishnan Rajamony, Weimin Yu, and Willy Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, February 1996.

[3] Thomas E. Anderson, David E. Culler, and David A. Patterson. A case for NOW (networks of workstations. *IEEE Micro*, 15(1):54–64, February 1995.

[4] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, 1992.

[5] Remzi H. Arpaci, Andrea C. Dusseau, Amin M. Vahdat, Lok T. Liu, Thomas E. Anderson, and David A. Patterson. The interaction of parallel and sequential workloads on a network of workstations. In *Proceedings of the 1995 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 267–278, May 1995.

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

[7] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. Dag-consistent distributed shared memory. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS)*, pages 132–141, Honolulu, Hawaii, April 1996.

[8] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[9] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 356–368, Santa Fe, New Mexico, November 1994.

[10] Robert D. Blumofe and David S. Park. Scheduling large-scale parallel computations on networks of workstations. In *Proceedings of the Third International Symposium on High Performance Distributed Computing (HPDC)*, pages 96–105, San Francisco, California, August 1994.

[11] Clemens H. Cap and Volker Strumpen. Efficient parallel computing in distributed workstation environments. *Parallel Computing*, 19:1221–1234, 1993.

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

[13] Vint Cerf and Robert Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Computers*, 22(5):637–648, May 1974.

[14] Rohit Chandra, Anoop Gupta, and John L. Hennessy. COOL: An object-based language for parallel programming. *IEEE Computer*, 27(8):13–26, August 1994.

[15] John Chapin, Mendel Rosenblum, Scott Devine, Tirthankar Lahiri, Dan Teodosiu, and Anoop Gupta. Hive: Fault containment for shared-memory multiprocessors. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP)*, pages 12–25, Copper Mountain Resort, Colorado, December 1995.

[16] Jeffrey S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. The Amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP 12)*, pages 147–158, Litchfield Park, Arizona, December 1989.

[17] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[18] Henry Clark and Bruce McMillin. DAWGS—a distributed compute server utilizing idle workstations. *Journal of Parallel and Distributed Computing*, 14(2):175–186, February 1992.

[19] P. Dasgupta, R. C. Chen, S. Menon, M. P. Pearson, R. Ananthanarayanan, U. Ramachandran, M. Ahamad, R. J. LeBlanc, W. F. Appelbe, J. M. Bernabéu-Aubán, P. W. Hutto, M. Y. A. Khalidi, and C. J. Wilkenloh. The design and implementation of the Clouds distributed operating system. *Computing Systems*, 3(1):11–46, 1990.

[20] Fred Douglis and John Ousterhout. Transparent process migration: Design alternatives and the sprite implementation. *Software—Practice and Experience*, 21(8):757–785, August 1991.

[21] Elmootazbellah N. Elnozahy and Willy Zwaenepoel. Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output commit. *IEEE Transactions on Computers*, C-41(5):526–531, May 1992.

[22] Robert E. Felderman, Eve M. Schooler, and Leonard Kleinrock. The Benevolent Bandit Laboratory: A testbed for distributed algorithms. *IEEE Journal on Selected Areas in Communications*, 7(2):303–311, February 1989.

[23] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Distributed Filaments: Efficient fine-grain parallelism on a cluster of workstations. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 201–213, Monterey, California, November 1994.

[24] David Gelernter and David Kaminsky. Supercomputing out of recycled garbage: Preliminary experience with Piranha. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, pages 417–427, Washington, D.C., July 1992.

[25] Chris Joerg and Bradley C. Kuszmaul. Massively parallel chess. In *Proceedings of the Third DIMACS Parallel Implementation Challenge*, Rutgers University, New Jersey, October 1994. Available as `ftp://theory.lcs.mit.edu/pub/cilk/dimacs94.ps.Z`.

[26] Christopher F. Joerg. *The Cilk System for Parallel Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January 1996.

[27] David Louis Kaminsky. *Adaptive Parallelism with Piranha*. PhD thesis, Yale University, May 1994.

[28] Phillip Krueger and Rohit Chawla. The Stealth distributed scheduler. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 336–343, Arlington, Texas, May 1991.

[29] Kai Li. IVY: A shared virtual memory system for parallel computing. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 94–101, August 1988.

[30] Philip Andrew Lisiecki. Macro-level scheduling in the Cilk network of workstations environment. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1996.

[31] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor—a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Comput-*

ing Systems, pages 104–111, San Jose, California, June 1988.

[32] Howard J. Lu. Heterogeneous multithreaded computing. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1995.

[33] 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, May 1995.

[34] Steven P. Miller, B. Clifford Neuman, Jeffrey I. Schiller, and Jermoe H. Saltzer. Kerberos authentication and authorization system. Athena technical plan, M.I.T. Project Athena, October 1988.

[35] David A. Nichols. Using idle workstations in a shared computing environment. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles (SOSP 11)*, pages 5–12, Austin, Texas, November 1987.

[36] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[37] Vijay S. Pande, Christopher F. Joerg, Alexander Yu Grosberg, and Toyoichi Tanaka. Enumerations of the hamiltonian walks on a cubic sublattice. *Journal of Physics A*, 27, 1994.

[38] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[39] Daniel J. Scales and Monica S. Lam. Transparent fault tolerance for parallel applications on networks of workstations. In *Proceedings of the USENIX 1996 Annual Winter Technical Conference*, San Diego, California, January 1996.

[40] W. Richard Stevens. *UNIX Network Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[41] V. S. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[42] Andrew S. Tanenbaum, Henri E. Bal, and M. Frans Kaashoek. Programming a distributed system using shared objects. In *Proceedings of the Second International Symposium on High Performance Distributed Computing*, pages 5–12, Spokane, Washington, July 1993.

[43] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–63, December 1990.

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

[45] Carl A. Waldspurger, Tad Hogg, Bernardo A. Huberman, Jeffrey O. Kephart, and W. Scott Stornetta. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, February 1992.

[46] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *Proceedings of the Ninth ACM Symposium on Operating Systems Principles (SOSP)*, pages 49–70, Bretton Woods, New Hampshire, October 1983.

[47] Songnian Zhou, Jingwen Wang, Xiaohu Zheng, and Pierre Delisle. Utopia: A load sharing facility for large, heterogeneous distributed computer systems. *Software—Practice and Experience*, 23(12):1305–1336, December 1993.