Dag-Consistent Distributed Shared Memory

Robert D. Blumofe Matteo Frigo Christopher F. Joerg
Charles E. Leiserson Keith H. Randall

MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139

Abstract
We introduce dag consistency, a relaxed consistency model for distributed shared memory which is suitable for multi-threaded programming. We have implemented dag consistency in software for the Cilk multithreaded runtime system running on a Connection Machine CM5. Our implementation includes a dag-consistent distributed cactus stack for storage allocation. We provide empirical evidence of the flexibility and efficiency of dag consistency for applications that include blocked matrix multiplication, Strassen’s matrix multiplication algorithm, and a Barnes-Hut code. Although Cilk schedules the executions of these programs dynamically, their performances are competitive with statically scheduled implementations in the literature. We also prove that the number $F_P$ of page faults incurred by a user program running on $P$ processors can be related to the number $F_1$ of page faults running serially by the formula $F_P \leq F_1 + 2Cs$, where $C$ is the cache size and $s$ is the number of thread migrations executed by Cilk’s scheduler.

1 Introduction
Architects of shared memory for parallel computers have attempted to support Lamport’s model of sequential consistency [22]: The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. Unfortunately, they have generally found that Lamport’s model is difficult to implement efficiently, and hence relaxed models of shared-memory consistency have been developed [10, 12, 13] that compromise on semantics for a faster implementation. By and large, all of these consistency models have had one thing in common: they are “processor centric” in the sense that they define consistency in terms of actions by physical processors. In this paper, we introduce “dag” consistency, a relaxed consistency model based on user-level threads which we have implemented for Cilk [4], a C-based multithreaded language and runtime system.

Dag consistency is defined on the dag of threads that make up a parallel computation. Intuitively, a read can “see” a write in the dag-consistency model only if there is some serial execution order consistent with the dag in which the read sees the write. Unlike sequential consistency, but similar to certain processor-centric models [12, 14], dag consistency allows different reads to return values that are based on different serial orders, but the values returned must respect the dependencies in the dag.

The current Cilk mechanisms to support dag-consistent distributed shared memory on the Connection Machine CM5 are implemented in software. Nevertheless, codes such as matrix multiplication run efficiently, as can be seen in Figure 1. The dag-consistent shared memory performs at 5 mega-ops per processor as long as the work per processor is sufficiently large. This performance compares fairly well with other matrix multiplication codes on the CM5 (that do not use the CM5’s vector units). For example, an implementation coded in Split-C [9] attains just over 6 mega-ops per processor on 64 processors using a static data layout, a static thread schedule, and an optimized assembly-language inner loop. In contrast, Cilk’s dag-consistent shared memory is mapped across the processors dynamically, and the Cilk threads performing the computation are scheduled dynamically at runtime. We believe that the overhead in our current implementation can be reduced, but that in any case, this overhead is a reasonable price to pay for ease of programming and dynamic load balancing.

We have implemented irregular applications that employ...
To illustrate the concepts behind dag consistency, consider the problem of parallel matrix multiplication. One way to program matrix multiplication is to use the recursive divide-and-conquer algorithm shown in Figure 2. To multiply one \( n \times n \) matrix by another, we divide each matrix into four \( n/2 \times n/2 \) submatrices, recursively compute some products of these submatrices, and then add the results together. This algorithm lends itself to a parallel implementation, because each of the eight recursive multiplications is independent and can be executed in parallel.

Figure 3 shows Cilk code\(^1\) for a “blocked” implementation of recursive matrix multiplication in which the (square) input matrices \( A \) and \( B \) and the output matrix \( R \) are stored as a collection of \( 16 \times 16 \) submatrices, called \( \textit{blocks} \). All three matrices are abstractly stored in Cilk’s shared memory, but the CM5 implementation distributes their elements among the individual processor memories. The Cilk procedure \texttt{matrixmul} takes as arguments pointers to the first block in each matrix as well as a variable \( \texttt{nb} \) denoting the number of blocks in any row or column of the matrices. From the pointer to the first block of a matrix and the value of \( \texttt{nb} \), the location of any other block in the matrix can be computed quickly. As \texttt{matrixmul} executes, values are stored into \( R \), as well as into a temporary matrix \( tmp \).

The procedure \texttt{matrixmul} operates as follows. Lines 3–4 check to see if the matrices to be multiplied consist of a single block, in which case a call is made to a serial routine \texttt{multiply-block} (not shown) to perform the multiplication. Otherwise, line 8 allocates some page-aligned temporary storage in shared memory for the results, lines 9–10 compute pointers to the 8 submatrices of \( A \) and \( B \), and lines 11–12 compute pointers to the 8 submatrices of \( R \) and the temporary matrix \( tmp \). At this point, the divide step of the divide-and-conquer paradigm is complete, and we begin on the conquer step. Lines 13–20 recursively compute the 8 required submatrix multiplications, storing the results in the 8 disjoint submatrices of \( R \) and \( tmp \). The recursion is made to execute in parallel by using the \texttt{spawn} directive, which is similar to a \texttt{C} function call except that the caller can continue to execute even if the callee has not yet returned. The \texttt{sync} statement in line 21 causes the procedure to suspend until all the procedures it spawned have finished.

\(^1\)Shown is Cilk-3 code, which provides explicit linguistic support for shared-memory operations and call/return semantics for coordinating threads. The original Cilk-1 system [4] used explicit continuation passing to coordinate threads. For a history of the evolution of Cilk, see [17].
1 cilk void matrixmul(long nb, shared block *A, 
shared block *B, 
shared block *R) 
2  
3    if (nb == 1) 
4    multiply_block(A, B, R); 
5  else { 
6      shared block *C,*D,*E,*F,*G,*H,*I,*J; 
7      shared block *CG,*CH,*EG,*EH, 
*DI,*DJ,*FI,*FJ; 
8      shared page_aligned block tmp[nb*nb]; 
9      /* get pointers to input submatrices */ 
10     partition(nb, A, &C, &D, &E, &F); 
11     partition(nb, B, &G, &H, &I, &J); 
12     /* get pointers to result submatrices */ 
13     partition(nb, R, &CG, &CH, &EG, &EH); 
14     partition(nb, tmp, &DI, &DJ, &FI, &FJ); 
15     /* solve subproblems recursively */ 
16     spawn matrixmul(nb/2, C, G, CG); 
17     spawn matrixmul(nb/2, C, H, CH); 
18     spawn matrixmul(nb/2, E, H, EH); 
19     spawn matrixmul(nb/2, E, G, EG); 
20     spawn matrixmul(nb/2, D, I, DI); 
21     spawn matrixmul(nb/2, D, J, DJ); 
22     spawn matrixmul(nb/2, F, J, FJ); 
23     spawn matrixmul(nb/2, F, I, FI); 
24     sync; 
25     /* add results together into R */ 
26     spawn matrixadd(nb, tmp, R); 
27     sync; 
28 } 
29 return; 
30 }

Figure 3: Cilk code for recursive blocked matrix multiplication.

(The sync statement is not a global barrier.) Then, line 22 spawns a parallel addition in which the matrix tmp is added into R. (The procedure matrixadd is itself implemented in a recursive, parallel, divide-and-conquer fashion, and the code is not shown.) The sync in line 23 ensures that the addition completes before matrixmul returns.

Like any Cilk multithreaded computation [4], the parallel instruction stream of matrixmul can be viewed as a “spawn tree” of procedures broken into a directed acyclic graph, or dag, of “threads.” The spawn tree is exactly analogous to a traditional call tree. When a procedure, such as matrixmul performs a spawn, the spawned procedure becomes a child of the procedure that performed the spawn. Each procedure is broken by sync statements into non-blocking sequences of instructions, called threads, and the threads of the computation are organized into a dag representing the partial execution order defined by the program. Figure 4 illustrates the structure of the dag for matrixmul. Each vertex corresponds to a thread of the computation, and the edges define the partial execution order. The syncs in lines 21 and 23 break the procedure matrixmul into subproblems $M_1, M_2, \ldots, M_8$ in lines 2–20, the spawning of the addition $S$ in line 22, and the return in line 25.

The Cilk runtime system automatically schedules the execution of the computation on the processors of a parallel computer using the provably efficient technique of work stealing [5], in which idle processors steal spawned procedures from victim processors chosen at random. When a procedure is stolen, we refer to it and all of its descendant procedures in the spawn tree as a subcomputation.

Dag-consistent shared memory is a natural consistency model to support a shared-memory program such as matrixmul. Certainly, sequential consistency can guarantee the correctness of the program, but a closer look at the precedence relation given by the dag reveals that a much weaker consistency model suffices. Specifically, the 8 recursively spawned children $M_1, M_2, \ldots, M_8$ need not have the same view of shared memory, because the portion of shared memory that each writes is neither read nor written by the others. On the other hand, the parallel addition of tmp into R by the computation S requires S to have a view in which all of the writes to shared memory by $M_1, M_2, \ldots, M_8$ have completed.

The intuition behind dag consistency is that each thread sees values that are consistent with some serial execution order of the dag, but two different threads may see different serial orders. Thus, the writes performed by a thread are seen by its successors, but threads that are incomparable in the dag may or may not see each other’s writes. In matrixmul, the computation $S$ sees the writes of $M_1, M_2, \ldots, M_8$, because all the threads of $S$ are successors of $M_1, M_2, \ldots, M_8$, but since the $M_i$ are incomparable, they cannot depend on seeing each others writes. We shall define dag consistency precisely in Section 3.

3 The BACKER coherence algorithm

This section describes our coherence algorithm, which we call BACKER, for maintaining dag consistency. We first give a formal definition of dag-consistent shared memory and explain how it relates to the intuition of dag consis-
tency that we have gained thus far. We then describe the cache and “backing store” used by BACKER to store shared-memory objects, and we give three fundamental operations for moving shared-memory objects between cache and backing store. Finally, we give the BACKER algorithm and describe how it ensures dag consistency.

We first introduce some terminology. Let \( G = (V, E) \) be the dag of a multithreaded computation. For \( i,j \in V \), if a path of nonzero length from thread \( i \) to thread \( j \) exists in \( G \), we say that \( i \) (strictly) precedes \( j \), which we write \( i \prec j \). We say that two threads \( i, j \in V \) with \( i \neq j \) are incomparable if we have \( i \not\prec j \) and \( j \not\prec i \).

Shared memory consists of a set of objects that threads can read and write. To track which thread is responsible for an object’s value, we imagine that each shared-memory object has a tag which the write operation sets to the name of the thread performing the write. We assume without loss of generality that each thread performs at most one read or write.

We define dag consistency as follows. (See also [17].)

**Definition 1** The shared memory \( M \) of a multithreaded computation \( G = (V, E) \) is dag-consistent if the following two conditions hold:

1. Whenever any thread \( i \in V \) reads any object \( m \in M \), it receives a value \( v \) tagged with some thread \( j \in V \) such that \( j \) writes \( v \) to \( m \) and \( i \not\prec j \).
2. For any three threads \( i, j, k \in V \), satisfying \( i \prec j \prec k \), if \( j \) writes some object \( m \in M \) and \( k \) reads \( m \), then the value received by \( k \) is not tagged with \( i \).

For deterministic programs, this definition implies the intuitive notion that a read can “see” a write only if there is some serial execution order of the dag in which the read sees the write. As it turns out, however, this intuition is ill defined for certain nondeterministic programs. For example, there exist nondeterministic programs whose parallel execution can contain reads that do not occur in any serial execution. Definition 1 implies the intuitive semantics for deterministic programs and is well defined for all programs.

Programs can easily be written that are guaranteed to be deterministic. Nondeterminism arises when a write to an object occurs that is incomparable with another read or write (of a different value) to the same object. For example, if a read and a write to the same object are incomparable, then the read might or might not receive the value of the write. Similarly, if two writes are incomparable and a read exists that succeeds them both with no other intervening writes, the read might receive the value of either write. To avoid nondeterminism, it suffices that no write to an object occurs that is incomparable with another read or write to the same object, in which case all writes to the object must lie on a single path in the dag. Moreover, all writes and any one given read must also lie on a single path. Consequently, by Definition 1, every read of an object sees exactly one write to that object, and the execution is deterministic.

We now describe the BACKER coherence algorithm for maintaining dag-consistent shared memory. In this algorithm, versions of shared-memory objects can reside simultaneously in any of the processors’ local caches or the global backing store. Each processor’s cache contains objects recently used by the threads that have executed on that processor, and the backing store provides default global storage for each object. For our Cilk system on the CM5, portions of each processor’s main memory are reserved for the processor’s cache and for a portion of the distributed backing store, although on some systems, it might be reasonable to implement the backing store on disk. In order for a thread executing on the processor to read or write an object, the object must be in the processor’s cache. Each object in the cache has a dirty bit to record whether the object has been modified since it was brought into the cache.

Three basic operations are used by the BACKER to manipulate shared-memory objects: fetch, reconcile, and flush. A fetch copies an object from the backing store to a processor cache and marks the cached object as clean. A reconcile copies a dirty object from a processor cache to the backing store and marks the cached object as clean. Finally, a flush removes a clean object from a processor cache. Unlike implementations of other models of consistency, all three operations are bilateral between a processor’s cache and the backing store, and other processors’ caches are never involved.

The BACKER coherence algorithm operates as follows. When the user code performs a read or write operation on an object, the operation is performed directly on a cached copy of the object. If the object is not in the cache, it is fetched from the backing store before the operation is performed. If the operation is a read, the dirty bit of the object is set. To make space in the cache for a new object, a clean object can be removed by flushing it from the cache. To remove a dirty object, it is reconciled and then flushed.

Besides performing these basic operations in response to user reads and writes, the BACKER performs additional reconciles and flushes to enforce dag consistency. For each edge \( i \rightarrow j \) in the computation dag, if threads \( i \) and \( j \) are scheduled on different processors, say \( p \) and \( q \), then BACKER reconciles all of \( p \)’s cached objects after \( p \) executes \( i \) but before \( p \) enables \( j \), and it reconciles and flushes \( q \)’s entire cache before \( q \) executes \( j \).

The key reason BACKER works is that it is always safe, at any point during the execution, for a processor \( p \) to reconcile an object or to flush a clean object. Suppose we arbitrarily insert a reconcile of an object into the computation

\[ \text{See [17] for details of a “lazier” coherence algorithm than BACKER based on climbing the spawn tree.} \]
performed by \( p \). Assuming that there is no other communication involving \( p \), if \( p \) later fetches the object that it previously reconciled, it will receive either the value that it wrote earlier or a value written by a thread \( i \) that is incomparable with the thread \( i' \) performing the read. In the first case, part 2 of the Definition 1 is satisfied by the semantics of ordinary serial execution. In the second case, the thread \( i \) that performed the write is incomparable with \( i' \), and thus part 2 of the definition holds as well. (Part 1 of the definition holds trivially.)

The **BACKER** algorithm uses this safety property to guarantee dag consistency even when there is communication. Suppose that a thread \( i \) resides on processor \( p \) with an edge to a thread \( j \) on processor \( q \). In this case, **BACKER** causes \( p \) to reconcile all its cached objects after executing \( i \) but before enabling \( j \), and it causes \( q \) to reconcile and flush its entire cache before executing \( j \). At this point, the state of \( q \)'s cache (empty) is the same as \( p \)'s if \( j \) had executed with \( i \) on processor \( p \), but a reconcile and flush had occurred between them. Consequently, **BACKER** ensures dag consistency.

With all the reconciles and flushes being performed by the **BACKER** algorithm, why should we expect it to be an efficient coherence algorithm? The main reason is that once a processor has fetched an object into its cache, the object never needs to be updated with external values or invalidated, unless communication involving that processor occurs to enforce a dependency in the dag. Consequently, the processor can run with the speed of a serial algorithm with no overheads. Moreover, in Cilk, communication to enforce dependencies does not occur often [4].

It is worth mentioning that **BACKER** actually supports stronger semantics than Definition 1 requires. In fact, Definition 1 allows certain semantic anomalies, but **BACKER** handles these situations in the intuitively correct way. We are currently attempting to characterize the semantics of **BACKER** fully.

## 4 Implementation

This section describes our implementation of dag-consistent shared memory for the Cilk runtime system running on the Connection Machine Model CM5 parallel supercomputer [24]. We also describe the Cilk language extensions for supporting shared-memory objects and the "diff" mechanism [20] for managing dirty bits. Finally, we discuss minor anomalies in atomicity that can occur when the size of the concrete objects supported by the shared-memory system is different from the abstract objects that the programmer manipulates.

The Cilk system on the CM5 supports concrete shared-memory objects of 32-bit words. All consistency operations are logically performed on a per-word basis. If the runtime system had to operate on every word independently, however, the system would be terribly inefficient. Since extra fetches and reconciles do not adversely affect the **BACKER** coherence algorithm, we implemented the familiar strategy of grouping objects into *pages* [16, Section 8.2], each of which is fetched or reconciled as a unit. Assuming that spatial locality exists when objects are accessed, grouping objects helps amortize the runtime system overhead.

Unfortunately, the CM5 operating system does not support handling of page faults by user-level code, and so we were forced to implement shared memory in a relatively expensive fashion. Specifically, in our CM5 implementation, shared memory is kept separate from other user memory, and special operations are required to operate on it. Most painfully, testing for page faults occurs explicitly in software, rather than implicitly in hardware. Our *cilk2c* type-checking preprocessor [27] alleviates some of the discomfort, but a transparent solution that uses hardware support for paging would be preferable. A minor advantage to the software approach we use, however, is that we can support full 64-bit addressing of shared memory on the 32-bit SPARC processors of the CM5 system.

Cilk’s language support makes it easy to express operations on shared memory. The user can declare *shared* pointers and can operate on these pointers with normal C operations, such as pointer arithmetic and dereferencing. The type-checking preprocessor automatically generates code to perform these operations. The user can also declare *shared* arrays which are allocated and deallocated automatically by the system. As an optimization, we also provide *register shared* pointers, which are a version of *shared* pointers that are optimized for multiple accesses to the same page. In our CM5 system, a *register shared* pointer dereference takes 4 instructions when it performs multiple accesses to within a single page, as compared to 1 instruction for an ordinary C-pointer dereference. Finally, Cilk provides a loophole mechanism to convert *shared* pointers to C pointers, allowing direct, fast operations on pages. This loophole mechanism puts the onus on the user, however, for ensuring that the pointer stays within a single page. In the near future, we hope to port Cilk to an architecture and operating system that allow user-level handling of page faults. On such a platform, no difference will exist between *shared* objects and their C equivalents, and detecting page faults will incur no software overhead.

An important issue we faced with the implementation of dag-consistent shared memory on the CM5 was how to keep track of which objects on a page have been written. The CM5 provides no direct hardware support to maintain dirty bits explicitly at the granularity of words. Rather than using dirty bits explicitly, Cilk uses a *diff* mechanism as is used in the Treadmarks system [20]. The diff mechanism computes the dirty bit for an object by comparing that object’s value with its value in a copy made at fetch time. Our implementation makes this copy only for pages loaded in read/write...
mode, thereby avoiding the overhead of copying for read-only pages. The diff mechanism imposes extra overhead on each reconcile, but it imposes no extra overhead on each access [35].

Dag consistency can suffer from atomicity anomalies when abstract objects that the programmer is reading and writing are larger than the concrete objects supported by the shared-memory system. For example, suppose the programmer is treating two 4-byte concrete objects as one 8-byte abstract object. If two incomparable threads each write the entire 8-byte object, the programmer might expect an 8-byte read of the structure by a common successor to receive one of the two 8-byte values written. The 8-byte read may nondeterministically receive 4 bytes of one value and 4 bytes of the other value, however, since the 8-byte read is really two 4-byte reads, and the consistency of the two halves is maintained separately. Fortunately, this problem can only occur if the abstract program is nondeterministic, that is, if the program is nondeterministic even when the abstract and concrete objects are the same size. When writing deterministic programs, the programmer need not worry about this atomicity problem.

As with other consistency models, including sequential consistency, atomicity anomalies can also occur when the programmer packs several abstract objects into a single system object. Fortunately, this problem can easily be avoided in the standard way by not packing together abstract objects that might be updated in parallel.

5 Memory allocation

Some means of allocating memory must be provided in any useful implementation of shared memory. We considered implementing general heap storage in the style of C’s malloc and free, but most of our immediate applications only require stack-like allocation for temporary variables and the like. Since Cilk procedures operate in a parallel tree-like fashion, however, we needed some kind of parallel stack. We settled on implementing a cactus-stack [15, 28, 32] allocator.

From the point of view of a single Cilk procedure, a cactus stack behaves much like an ordinary stack. The procedure can allocate and free memory by incrementing and decrementing a stack pointer. The procedure views the stack as a linearly addressed space extending back from its own stack frame to the frame of its parent and continuing to more distant ancestors.

The stack becomes a cactus stack when multiple procedures execute in parallel, each with its own view of the stack that corresponds to its call history, as shown in Figure 5. In the figure, subcomputation $S_1$ allocates some memory $A$ before procedure $P_1$ is spawned. Subcomputation $S_1$ then continues to allocate more memory $B$. When procedure $P_1$ is stolen and becomes the root of subcomputation $S_2$, a new branch of the stack is started so that subsequent allocations performed by $S_2$ do not interfere with the stack being used by $S_1$. The stacks as seen by $S_1$ and $S_2$ are independent below the steal point, but they are identical above the steal point. Similarly, when procedure $P_2$ is stolen from $S_2$ to start subcomputation $S_3$, the cactus stack branches again.

Cactus-stack allocation mirrors the advantages of an ordinary procedure stack. Any object on the stack that is viewable by a procedure has a simple address: its offset from the base of the stack. Procedure local variables and arrays can be allocated and deallocated automatically by the runtime system in a natural fashion, as can be seen in the matrix multiplication example from Figure 3. Allocation can be performed completely locally without communication by simply incrementing a local pointer, although communication may be required when an out-of-cache stack page is actually referenced. Separate branches of the cactus stack are insulated from each other, allowing two subcomputations to allocate and free objects independently, even though objects may be allocated with the same address. Procedures can reference common data through the shared portion of their stack address space.

Cactus stacks have many of the same limitations as ordinary procedure stacks [28]. For instance, a child thread cannot return to its parent a pointer to an object that it has allocated. Similarly, sibling procedures cannot share storage that they create on the stack. Just as with a procedure stack, pointers to objects allocated on the cactus-stack can only be safely passed to procedures below the allocation point in the call tree. Heap storage offers a way of alleviating some of these limitations (and we intend to provide a heap alloca-
tor in a future version of Cilk), but the cactus stack provides simple and efficient support for allocation of procedure local variables and arrays.

The size of the backing store determines how large a shared-memory application one can run. On the CM5, the backing store is implemented in a distributed fashion by allocating a large fraction of each processor’s memory to this function. To determine which processor holds the backing store for a page, a hash function is applied to the page identifier (a pair of the cactus-stack address and the allocating subcomputation). A fetch or reconcile request for a page is made to the backing store of the processor to which the page hashes. This policy ensures that backing store is spread evenly across the processors’ memory. In other systems, it might be reasonable to place the backing store on disk à la traditional virtual memory.

6 Analysis of Page Faults

In this section, we examine the number of page faults of a Cilk computation when run on \( P \) processors using Cilk’s randomized work-stealing scheduler [4] and the implementation of the ACKER coherence algorithm described in Section 4. We prove that \( F_P \) can be related to the number \( F_1 \) of page faults taken by a 1-processor execution by the formula \( F_P \leq F_1 + 2Cs \), where \( C \) is the size of each processor’s cache in pages and \( s \) is the total number of steals executed by Cilk’s scheduler. The \( 2Cs \) term represents faults due to “warming up” the processors’ caches, and we present empirical evidence that this overhead is actually much smaller in practice than the theoretical bound.

We begin with a theorem that bounds the number of page faults of a Cilk application. The proof takes advantage of properties of the least-recently used (LRU) page replacement scheme used by Cilk, as well as the fact that Cilk’s scheduler, like C, executes serial code in a depth-first fashion.

**Theorem 1** Let \( F_P \) be the number of page faults of a Cilk computation when run on \( P \) processors with a cache of \( C \) pages on each processor. Then, we have \( F_P \leq F_1 + 2Cs \), where \( s \) is the total number of steals that occur during Cilk’s execution of the computation.

**Proof:** The proof is by induction on the number \( s \) of steals. For the base case, observe that if no steals occur, then the application runs entirely on one processor, and thus it faults \( F_1 \) times by definition. For the inductive case, consider an execution \( E \) of the computation that has \( s \) steals. Choose any subcomputation \( T \) from which no processor steals during the execution \( E \). Construct a new execution \( E' \) of the computation which is identical to \( E \), except that \( T \) is never stolen. Since \( E' \) has only \( s-1 \) steals, we know it has at most \( F_1 + 2C(s - 1) \) page faults by the inductive hypothesis.

To relate the number of page faults during execution \( E \) to the number during execution \( E' \), we examine cache behavior under LRU replacement. Consider two processors that execute simultaneously and in lock step a block of code using two different starting cache states, where each processor’s cache has \( C \) pages. The main property of LRU that we exploit is that the number of page faults in the two executions can differ by at most \( C \) page faults. This property follows from the observation that no matter what the starting cache states might be, the states of the two caches must be identical after one of the two executions takes \( C \) page faults. Indeed, at the point when one execution has just taken its \( C \)th page fault, each cache contains exactly the last \( C \) distinct pages referenced [19].

We can now count the number of page faults during the execution \( E \). The fault behavior of \( E \) is the same as the fault behavior of \( E' \) except for the subcomputation \( T \) and the subcomputation, call it \( U \), from which it stole. Since \( T \) is executed in depth-first fashion, the only difference between the two executions is that the starting cache state of \( T \) and the starting cache state of \( U \) after \( T \) are different. Therefore, execution \( E \) makes at most \( 2C \) more page faults than execution \( E' \), and thus execution \( E \) has at most \( F_1 + 2C(s - 1) + 2C = F_1 + 2Cs \) page faults.

Theorem 1 says that the total number of faults on \( P \) processors is at most the total number of faults on 1 processor plus an overhead term. The overhead arises whenever a steal occurs, because in the worst case, the caches of both the theiving processor and the victim processor contain no pages in common compared to the situation when the steal did not occur. Thus, they must be “warmed up” until the caches “synchronize” with the cache of a serial execution.

To measure the warm-up overhead, we counted the number of page faults taken by several applications—including matrixmul, an optimized matrix multiplication routine, and a parallel version of Strassen’s algorithm [33]—for various choices of cache, processor, and problem size. For each run we measured the cache warm-up fraction \( (F_P - F_1)/2Cs \), which represents the fraction of the cache that needs to be warmed up on each steal. We know from Theorem 1 that the cache warm-up fraction is at most 1. Our experiments indicate that the cache warm-up fraction is, in fact, typically less than 3%, as can be seen from the histogram in Figure 6 showing the cache warm-up fraction for 153 experimental runs of the above applications, with processor counts ranging from 2 to 64 and cache sizes from 256KB to 2MB. Thus, we see less than 3% of the extra \( 2Cs \) faults.

To understand why cache warm-up costs are so low, we performed an experiment that recorded the size of each subproblem stolen. We observed that most of the subproblems stolen during an execution were small. In fact, only 5–10%
of the stolen subproblems were “large,” where a large subproblem is defined to be one that takes \( C \) or more pages to execute. The other 90–95\% of the subproblems are small and are stolen when little work is left to do and many of the processors are idle. Therefore, most of the stolen subproblems never perform \( C \) page faults before terminating. The bound \( F_P \leq F_1 + 2C_s \) derived in Theorem 1 thus appears to be rather loose, and our experiments indicate that much better performance can be expected.

7 Performance

In this section, we model the performance of Cilk on synthetic benchmark applications similar to matrixmul. We quantify performance in terms of “work” and “critical-path length.” The work \( T_1 \) of a computation is the running time, including page faults, of the computation on one processor, when the backing store is running on other processors. The critical-path length \( T_\infty \) is the (theoretical) running time on an infinite number of processors assuming that page faults take zero time. Their ratio \( T_1/T_\infty \) is the average parallelism of the computation. We found that the running time \( T_P \) of the benchmarks on \( P \) processors can be estimated as \( T_P \approx 1.34(T_1/P) + 5.1(T_\infty) \). Speedup was always at least a third of perfect linear speedup for benchmarks with large average parallelism and running time was always within a factor of 10 of optimal for those with much parallelism.

To analyze Cilk’s implementation of the BACKER coherence algorithm, we measured the work and critical-path length for synthetic benchmarks obtained by adding \texttt{sync} statements to the matrix multiplication program shown in Figure 3. By judiciously placing \texttt{sync} statements in the code, we were able to obtain synthetic benchmarks that exhibited a wide range of average parallelism. We ran the benchmarks on various numbers of processors, each time recording the number \( P \) of processors and the actual runtime \( T_P \).

Figure 7 shows a normalized speedup curve [4] for the synthetic benchmarks. This curve is obtained by plotting speedup \( T_1/T_P \) versus machine size \( P \), but normalizing each of these values by dividing them by the average parallelism \( T_1/T_\infty \). We use a normalized speedup curve, because it allows us to plot runs of different benchmarks on the same graph. Also plotted in the figure are the perfect linear-speedup curve \( T_P = T_1/P \) (the 45\(^\circ\) line) and the limit on performance given by the parallelism bound \( T_P \geq T_\infty \) (the horizontal line).

The quantity \( T_\infty \) is not necessarily a tight lower bound on \( T_P \), because it ignores page faults. Indeed, the structure of matrixmul on \( n \times n \) matrices causes \( \Omega(lg n) \) faults to be taken along any path through the dag. A better measure, which we shall denote \( T_\infty(C) \), is the maximum, over all paths in the dag, of the time (including page faults) to execute all threads along the path on one processor with a cache size of \( C \) pages. Although the bound \( T_P \geq T_\infty(C) \) is tighter (and makes our numbers look better), it appears difficult to compute. We can estimate using analytical techniques, however, that for our matrix multiplication algorithms, \( T_\infty(C) \) is about twice as large as \( T_\infty \). Had we used this value for \( T_\infty \) in the normalized speedup curve in Figure 7, each data point would shift up and right by this factor of 2, giving somewhat tighter results.

The normalized speedup curve in Figure 7 shows that dag-consistent shared-memory applications can obtain good speedups. The data was fit to a curve of the form \( T_P = c_1 T_1/P + c_\infty T_\infty \). We obtained a fit with \( c_1 = 1.34 \) and \( c_\infty = 5.1 \), with an \( R^2 \) correlation coefficient of 0.963 and a mean relative error of 13.8\%. Thus, the shared memory
imposes about a 34% performance penalty on the work of an algorithm, and a factor of 5 performance penalty on the critical path. The factor of 5 on the critical path term is quite good considering all of the scheduling, protocol, and communication that could potentially contribute to this term.

There are two possible explanations for the additional 34% on the work term. The extra work could represent congestion at the backing store, which causes page faults to cost more than in the one-processor run. Alternatively, it could be because our $T_1$ measure is too conservative. To compute $T_1$, we run the backing store on processors other than the one running the benchmark, while when we run on $P$ processors, we use the same $P$ processors to implement the backing store. We have not yet run experiments to see which of these two explanations is correct.

8 Conclusion

Many other researchers have investigated distributed shared memory. To conclude, we briefly outline work in this area and offer some ideas for the future.

The notion that independent tasks may have incoherent views of each others’ memory is not new to Cilk. The BLAZE [26] language incorporated a memory semantics similar to that of dag consistency into a PASCAL-like language. The Myrias [2] computer was designed to support a relaxed memory semantics similar to dag consistency, with many of the mechanisms implemented in hardware. Loosely-Coherent Memory [23] allows for a range of consistency protocols and uses compiler support to direct their use. Compared with these systems, Cilk provides a multithreaded programming model based on directed acyclic graphs, which leads to a more flexible linguistic expression of operations on shared memory.

Cilk’s implementation of dag consistency borrows heavily on the experiences from previous implementations of distributed shared memory. Like Ivy [25] and others [6, 11, 20], Cilk’s implementation uses fixed-sized pages to cut down on the overhead of managing shared objects. In contrast, systems that use cache lines [7, 21, 29] require some degree of hardware support [31] to manage shared memory efficiently. As another alternative, systems that use arbitrary-sized objects or regions [8, 18, 30, 34] require either an object-oriented programming model or explicit user management of objects.

The idea of dag-consistent shared memory can be extended to the domain of file I/O to allow multiple threads to read and write the same file in parallel. We anticipate that it should be possible to memory-map files and use our existing dag-consistency mechanisms to provide a parallel, asynchronous, I/O capability for Cilk.

We are also currently working on porting dag-consistent shared memory to Cilk-NOW [3] adaptively parallel, fault-tolerant, network-of-workstations system. We are using operating system hooks to make the use of shared memory be transparent to the user. We expect that the well-structured nature of Cilk computations will allow the runtime system to maintain dag consistency efficiently, even in the presence of processor faults.

Acknowledgments

We gratefully acknowledge the work of Bradley Kuszmaul of Yale, Yuli Zhou of AT&T Bell Laboratories, and Rob Miller of Carnegie Mellon, all formerly of MIT, for their efforts as part of the Cilk team. Rob implemented the type-checking preprocessor for Cilk and led and implemented the design of the linguistics for explicit shared memory. Yuli undertook the first port of the $N$-body code to Cilk. Thanks to Burton Smith of Tera Computer Corporation for acquainting us with related work. Thanks to Mingdong Feng of the National University of Singapore for porting Cilk to the IBM SP-2 and providing feedback on our research. Thanks to the National University of Singapore for resources used to prepare the final version of this paper. Thanks to Arvind and his dataflow group at MIT for helpful discussions and inspiration.

References


