Modular Competitiveness for Distributed Algorithms

James Aspnes* Orli Waarts†

Abstract

We define a novel measure of competitive performance for distributed algorithms based on *throughput*, the number of tasks that an algorithm can carry out in a fixed amount of work. An important property of the throughput measure is that it is modular: we define a notion of *relative competitiveness* with the property that a k-relatively competitive implementation of an object T using a subroutine U, combined with an l-competitive implementation of U, gives a kl-competitive algorithm for T. We prove the throughput-competitiveness of an algorithm for a fundamental building block of many well-known distributed algorithms: the cooperative collect primitive. This permits a straightforward construction of competitive versions of these algorithms— the first examples of algorithms obtained through a general method for modular construction of competitive distributed algorithms. Moreover, we provide a lower bound that shows that the throughput competitiveness of the cooperative collect algorithm we study is nearly optimal.

Thus, we see our paper as making two main contributions: one is the introduction of a modular measurement for competitiveness, whose interest is justified by the throughput competitiveness of the cooperative collect algorithms; and the other is a technique for proving throughput competitiveness, which may apply to other distributed problems.

*Yale University, Department of Computer Science, 51 Prospect Street/P.O. Box 208285, New Haven CT 06520-8285. Supported by NSF grants CCR-9410228 and CCR-9415410. E-mail: aspnes-james@cs.yale.edu

†Computer Science Division, U. C. Berkeley. Supported in part by an NSF postdoctoral fellowship. E-Mail: waarts@cs.berkeley.edu

1 Introduction

Competitive analysis and distributed algorithms. The tool of competitive analysis was proposed by Sleator and Tarjan [50] to study problems that arise in an *on-line* setting, where an algorithm is given an unpredictable sequence of requests to perform operations, and must make decisions about how to satisfy its current request that may affect how efficiently it can satisfy future requests. Since the worst-case performance of an algorithm might depend only on very unusual or artificial sequences of requests, or might even be unbounded if one allows arbitrary request sequences, one would like to look instead at how well the algorithm performs relative to some measure of difficulty for the request sequence. The key innovation of Sleator and Tarjan was to use as a measure of difficulty the performance of an optimal *off-line* algorithm, one allowed to see the entire request sequence before making any decisions about how to satisfy it. They defined the *competitive ratio*, which is the supremum, over all possible input sequences σ, of the ratio of the performance achieved by the on-line algorithm on σ to the performance achieved by the optimal off-line algorithm on σ, where the measure of performance depends on the particular problem.

In a distributed setting there are additional sources of nondeterminism other than the request sequence. These include process step times, request arrival times, message delivery times (in a message-passing system) and failures. Moreover, a distributed algorithm has to deal not only with the problems of lack of knowledge of future requests and future system behavior, but also with incomplete information about the *current* system state. Due to the additional type of nondeterminism in the distributed setting, it is not obvious how to extend the notion of competitive analysis to this environment.

Awerbuch, Kutten, and Peleg [17], and Bartal, Fiat, and Rabani [19] took the first steps in this direction. Their work was in the context of job scheduling and data management. In these papers, and in subsequent work [4, 14, 15, 21], the cost of a distributed on-line algorithm[1] is compared to the cost of an optimal *global-*

[1]Because most distributed algorithms have an on-line flavor, we use the terms *distributed algorithm* and *distributed on-line algorithm* interchangeably.

control algorithm. (This is also done implicitly in the earlier work of Awerbuch and Peleg [18].) As has been observed elsewhere (see, *e.g.* [15], paraphrased here), this imposes an additional handicap on the distributed on-line algorithm in comparison to the optimal algorithm: In the distributed algorithm the decisions are made based solely on local information. It is thus up to the algorithm to learn (at a price) the relevant part of the global state necessary to make a decision. The additional handicap imposed on the on-line distributed algorithm is that it is evaluated against the off-line algorithm that does *not* pay for overhead of control needed to make an intelligent decision.

Ajtai, Aspnes, Dwork, and Waarts [3] argued that in some cases a more refined measure is necessary, and that to achieve this the handicap of incomplete system information should be imposed not only on the distributed on-line algorithm but also on the optimal algorithm with which the on-line algorithm is compared. Otherwise, two distributed on-line algorithms may seem to have the same competitive ratio, while in fact one of them totally outperforms the other. Their approach is ultimately based on the observation that the purpose of competitive analysis for on-line algorithms is to allow comparison between *on-line* algorithms; the fictitious off-line algorithm is merely a means to this end. Therefore, the natural extension of competitiveness to distributed algorithms is to define a distributed algorithm as k-competitive if it never performs more than k times worse than any other *distributed* algorithm.

Cooperative Collect. To demonstrate the different notions of competitiveness, we study the problem of having processes repeatedly collect values by the *cooperative collect* primitive, first abstracted by Saks, Shavit, and Woll [49], described below.[2]

We assume the standard model for asynchronous shared-memory computation, in which n processes communicate by reading and writing to a set of single-writer n-reader *registers*. (We confine ourselves to single-writer registers because the construction of registers that can be written to by more than one process is one of the principal uses for the cooperative collect primitive.) As usual, a *step* is a read or a write to a shared variable. The algorithms are required to be *wait-free*: there is an *a priori* bound on the number of steps a process must take in order to satisfy a request, independent of the behavior of the other processes.

Many algorithms in the wait-free shared-memory model have an underlying structure in which processes repeatedly collect values using the *cooperative collect* primitive. In the *cooperative collect* primitive, processes perform the *collect* operation – an operation in which the process learns the values of a set of n registers, with the guarantee that each value learned is *fresh*: it was present in the register at some point during the collect. If each process reads every register, then this condition is trivially satisfied; however, this algorithm will perform a lot of redundant work when there is high concurrency. Interestingly, this is the (trivial) solution that is used in current literature on wait-free shared-memory applications in the standard shared memory model, including nearly all algorithms known to us for consensus, snapshots, coin flipping, bounded round numbers, timestamps, and multi-writer registers [1, 2, 5, 6, 7, 8, 9, 11, 13, 20, 22, 23, 24, 25, 28, 29, 32, 34, 35, 38, 39, 40, 42, 51].[3]

We assume that the *schedule* – which processes take steps at which times, and when the registers are updated – is under the control of an adversary. Intuitively, if all n processes are attempting to perform collect operations concurrently, the work can be partitioned so that each process performs significantly fewer than n reads. Thus, more sophisticated protocols may allow one process p to learn values indirectly from another process q. Nevertheless, the worst-case cost for any distributed algorithm is always as high as the cost of the naïve algorithm, as follows. Suppose p_1 performs a collect in isolation. If p_2 later performs a collect, it cannot use the values obtained by p_1, since they might not be fresh. For this reason p_2 must read all the registers itself. Continuing this way, we can construct an execution in which every algorithm must have each process read all n registers. Thus, the worst-case cost for any distributed algorithm is always as high as the cost of the naïve algorithm.

The example shows that a worst-case measure is not very useful for evaluating cooperative collect algorithms. A similar example shows that a competitive analysis that proceeds by comparing a distributed algorithm to an ideal global-control algorithm gives equally poor results. The underlying difficulty arises because a global-control algorithm knows when registers are updated. Thus in the case where none of the registers have changed since a process's last collect, it can simply return the values it previously saw, doing *no* read or write operations. On the other hand, any distributed algorithm must read all n registers to be sure that new values have not appeared, which gives an infinite competitive ratio, for *any* distributed algorithm. Thus the competitive measure of [17, 19] does not allow one to distinguish between the naïve algorithm and algorithms that totally dominate it.

Competitive Latency. Observing the above was what led [3] to define a competitive measure for distributed algorithms, called *latency competitiveness*. The competitiveness presented in [3] allows such a distinction, *i.e.* between the naïve cooperative collect algorithm and algorithms that dominate it. To characterize the behavior of an algorithm over a range of possible schedules they define the *competitive latency* of an algorithm. Intuitively, the competitive latency measures the ratio between the amount of work that an algorithm needs to perform in order to carry out a particular set of collects, to the work done by the best possible algo-

[2] Much of this discussion is taken from [3].

[3] [49, 48] present interesting collect algorithms that do not follow the pattern of the naive algorithm. Both works however consider considerably stronger models than the standard shared memory model considered here.

rithm (*champion*) for carrying out those collects given the same schedule. In their model the schedule includes the timing of both system events and user requests. (See Figure 1.) As discussed above, they refine previous notions by requiring that this best possible algorithm be a distributed algorithm. Though the choice of this *champion* algorithm can depend on the schedule, and thus it can implicitly use its knowledge of the schedule to optimize performance (say, by having a process read a register that contains many needed values), it cannot cut corners that would compromise safety guarantees if the schedule were different (as it would if it allowed a process not to read a register because it "knows" from the schedule that the register has never been written to).

They then present the first (and so far the only) competitive algorithms for the cooperative collect problem. The basic technique in their algorithms is a mechanism that allows processes to read registers cooperatively, by having each process read registers in an order determined by a fixed permutation of the registers. They define the *collective latency* of an algorithm to be the worst case number of steps required to complete a set of collects that are in progress at a certain time. Using the trivial collect algorithm, even if n processes perform collects concurrently, there are a total of n^2 reads, *i.e.* the collective latency of the trivial algorithm is $O(n^2)$. [3] presents the first algorithms that cross this barrier. Their principal technical achievement is providing these nontrivial bounds on the collective latency of their algorithms. The techniques they present offer rather deep insight into the combinatorial structure of the problem and may be of more general use.

Competitive analysis and modularity. The Ajtai et al. [3] approach was successful: it distinguishes between the naive algorithm and faster cooperative collect algorithms; moreover, they provide such faster cooperative collect algorithms. Since collects appear either explicitly or implicitly in a wide variety of wait-free shared-memory algorithms, intuitively, a competitive collect protocol would translate into faster versions of these algorithms. However, competitive analysis in general appears to forbid a modular construction of competitive algorithms. If A is an algorithm that uses a subroutine B, the fact that B is competitive says nothing at all about A's competitiveness, since A must compete against algorithms that do not use B. Clearly, this lack of modularity impedes the development of practical competitive algorithms.

1.1 Our results

This paper overcomes this difficulty.

Competitive throughput. We define a new measure of competitive performance for distributed algorithms, called *competitive throughput*, that for the first time permits a general method for the modular construction of competitive distributed algorithms. *Throughput* measures the number of tasks carried out by an algorithm given a particular schedule. (See Figure 2.) The observation is that when analyzing a distributed algorithm it may be helpful to distinguish between two sources of nondeterminism, user requests (the input) and system behavior (the schedule). The work that compares a distributed algorithm with global control algorithm [4, 14, 15, 17, 18, 19, 21] implicitly makes this distinction by having the on-line and off-line algorithms compete only on the same input, generally hiding the details of the schedule in a worst-case assumption applied only to the on-line algorithm. The competitive latency model of [3] applies the same input and schedule to both the on-line and the off-line algorithms. In the present work the key insight is to preserve the split between the input and the schedule, as implicitly done in [4, 14, 15, 17, 18, 19, 21], but to reverse the approach of this previous work by assuming a worst-case *input* but a competitive *schedule*. That is, when comparing the number of tasks performed by a candidate algorithm with those performed by an optimal champion, we will assume that both are distributed algorithms that must deal with the same pattern of failures and asynchrony, but that the user requests given to the candidate are chosen to *minimize* the candidate's performance while the requests given to the champion are chosen to *maximize* the champion's performance.

Note that when comparing steps done by the candidate algorithm and champion algorithms, we consider the standard shared memory model for asynchronous shared memory computations, in which each step is an atomic read or write of shared memory. As explained above, this model is a natural model for the collect primitive; moreover, it has a simple mathematical structure and hence does not obscure the issues introduced in the paper. There is nothing in our notion of throughput competitiveness however that prohibits considering it in the context of more detailed models for multiprocessor architectures, such as one that takes into account the issue of contention (*i.e.* the number of processes trying to access the same variable concurrently). To incorporate contention one could simply make a process incur a stall step instead of a usual step in case of contention, following the lines of [30]. Note however that in the standard model considered in this paper, where a step is an atomic read or write to a single-writer n-reader register, the issue of contention is of lesser effect.

Relative Competitiveness. An important property of competitive throughput is that it allows competitive algorithms to be constructed modularly. We define a notion of *relative competitiveness* such that if A is a k-relative-competitive algorithm that calls an l-competitive subroutine B, then the combined algorithm $A \circ B$ is kl-competitive.

Competitive throughput is a natural measure to apply when the specific input has little effect on the number of tasks that can be completed, and most of the variation between executions is due to the schedule. This is often the case for shared-memory distributed

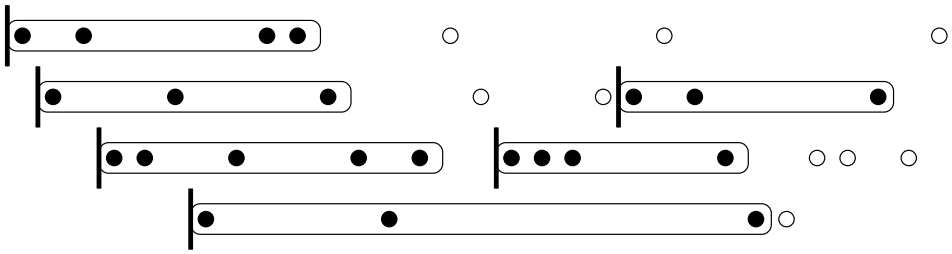

Figure 1: Latency model. New high-level operations (ovals) start at times specified by the scheduler (vertical bars). Scheduler also specifies timing of low-level operations (small circles). Cost to algorithm is number of low-level operations actually performed (filled circles).

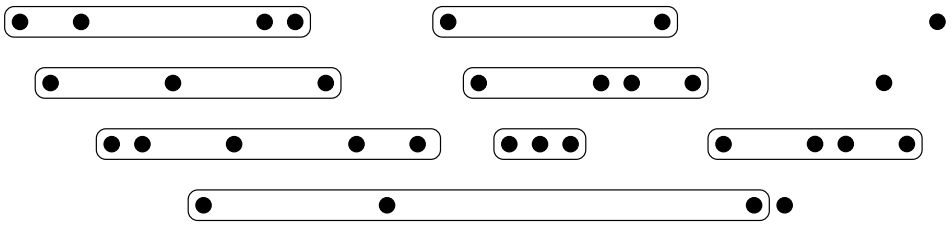

Figure 2: Throughput model. New high-level operations (ovals) start as soon as previous operations end. Scheduler controls only timing of low-level operations (filled circles). Payoff to algorithm is number of high-level operations completed.

algorithms.[4] Nevertheless, we do not view the notion of throughput competitiveness as replacing the notion of latency competitiveness. Competitive latency has the advantage of being more refined since the on-line and off-line algorithms are competing both on the schedule and on the input, *i.e.* have the same schedule and input. However, the competitive latency model, because it unifies the schedule and the input, seems unsuited to the modular construction of competitive algorithms. (A naive application of the latency model would only allow a candidate algorithm A to be compared against a champion that not only uses the same subroutine B but provides it with exactly the same requests arriving at exactly the same times.)

Cooperative Collect. As mentioned above, the principal technical difficulty of Ajtai et al. [3] was bounding the collective latency of their cooperative collects. They used this bound in order to provide a bound on the competitive latency of the cooperative collects. To use a bound on collective latency in order to bound the throughput competitiveness, one needs new, nontrivial ideas. Doing this is the principal technical achievement of our paper. In particular, we show that any collect algorithm with certain natural properties can be extended to a throughput-competitive implementation of a slightly stronger primitive, a *write-collect*.

For the fastest algorithm of [3] our result gives a throughput competitive ratio of $O(n^{3/4}\log^2 n)$. This is high, but the algorithms of [3] are the only ones that cross the trivial bound of $O(n)$ and hence the fastest current algorithms available. Moreover we show that they are nearly optimal: no cooperative collect algorithm can obtain a throughput competitiveness better than $\Omega(\sqrt{n})$. This high lower bound may indicate that an interesting research avenue may be to refine the notion of throughput competitiveness to reflect more accurately the performance of distributed algorithms. Currently throughput competitiveness, introduced here, is the most refined measure for evaluating throughput of distributed algorithms: the standard worst case measure and the approach of comparing a distributed algorithm with an ideal global control algorithm, will both give $\Omega(n)$ performance; and as described above the measure of [3] does not allow modularity.

It is the modularity property of throughput competitiveness that allows us to derive competitive versions of well-known shared-memory algorithms that are structured around the collect primitive, among which are the fastest known atomic snapshot algorithm, due to Attiya and Rachman [13], and the bounded round numbers abstraction due to Dwork, Herlihy, and Waarts [29]. An important consequence of the modularity of the throughput-competitive measure is that better algorithms for doing collects will immediately give better algorithms that use collects.

Thus, we see our paper as making two main contributions: one is the introduction of a modular measure-

[4] Where it is not (for example, when implementing an object that provides both cheap and expensive operations) we can often adjust the description of the problem to fit the measure. One way to do this is to join cheap operations to succeeding expensive operations. An example of this technique is given by the write-collect primitive described in Section 5.

ment for competitiveness, whose interest is justified by the throughput competitiveness of the cooperative collect algorithms; and the other is a technique for proving throughput competitiveness, which may apply to other distributed problems.

Due to lack of space, proofs are omitted or only sketched. A full version including detailed proofs is available [10].

1.2 Other Related Work

A notion related to allowing only correct distributed algorithms as champions is the very nice idea of comparing algorithms with partial information only against other algorithms with partial information. This was introduced by Papadimitriou and Yannakakis [46] in the context of linear programming; their model corresponds to a distributed system with no communication. A generalization of this approach has recently been described by Koutsoupias and Papadimitriou [41].

In addition, there is a long history of interest in *optimality* of a distributed algorithm given certain conditions, such as a particular pattern of failures [26, 31, 37, 43, 44, 45], or a particular pattern of message delivery [12, 33, 47]. In a sense, work on optimality envisions a fundamentally different role for the adversary in which it is trying to produce bad performance both in the candidate algorithm and in what we would call the champion algorithm; in contrast, the adversary used in competitive analysis usually cooperates with the champion.

Nothing in the literature corresponds in generality to our notion of relative competitiveness (Definition 2) and the Composition Theorem (Theorem 3) that uses it. Some examples of elegant specialized constructions of competitive algorithms from other competitive algorithms in a distributed setting are the *natural potential function* construction of Bartal, Fiat, and Rabani [19] and the distributed paging work of Awerbuch, Bartal, and Fiat [16]. However, not only do these constructions depend very much on the particular details of the problems being solved, but in addition they permit no concurrency, *i.e.* they assume that no two operations are ever in progress at the same time. (This assumption does not hold in general in typical distributed systems.) In contrast, the present work both introduces a general construction of modular competitive distributed algorithms *and* does so in the natural distributed setting that permits concurrency.

1.3 Possible Extensions

Our work defines modular competitiveness and relative competitiveness by distinguishing between two sources of nondeterminism, one of which is shared between the on-line and off-line algorithms, *i.e.* the schedule, and the other is not, *i.e.* the input. One can define analogous notions to modular competitiveness and to relative competitiveness by considering any two sources of nondeterminism, one of which is shared between the on-line and off-line algorithms, and one that is not. For example, we can obtain a notion of modular optimality, relative optimality, and a composition theorem for optimality, as follows: Let the shared nondeterminism correspond to observable properties of a system and let the distinct nondeterminism correspond to hidden properties. Then replace the best-case assumption for the off-line algorithm's input with a worst-case assumption. All of our definitions and results concerning modularity, including the composition theorem, will carry through, provided the replacement is done consistently. This is an interesting possible extension of this work that would be worth pursuing.

2 The Model

We assume the standard model for asynchronous shared-memory computation, in which n processes communicate by reading and writing to a set of single-writer n-reader atomic registers. Time proceeds in discrete units, in each of which some process takes a single atomic *step*. As usual, a step is a read or a write to a shared variable. The timing of events in the system is assumed to be under the control of an adversary, who is allowed to see the entire state of the system (including the internal states of the processes). The adversary decides at each time unit which process gets to take the next step; these decisions are summarized in the *schedule*, which formally is just a sequence of process id's. We require our algorithms to be *wait-free*: there is an *a priori* bound on the number of steps a process must take in order to satisfy a request, independent of the behavior of the other processes.

Most architectures provide some primitives that are stronger than atomic registers, such as Read-Modify-Write operation in which a register can be read and written in a single atomic step, and there are good reasons to study what can be done given such primitives. Nevertheless, the standard setting of atomic registers is well motivated. Among other reasons, stronger primitives are typically much more expensive than a read of an atomic register because of interactions with the cache hierarchy. When a process needs to perform an atomic read, it can often find the value in its cache (a *cache hit*), and hence does not need to actually go to the shared memory. In contrast, each time a process initiates a Read-Modify-Write operation, it must load from the shared memory.

We continue with the description of the model. Processes are always assumed to be carrying out tasks from a *request sequence* provided by the adversary. The request sequence corresponds to the inputs to the n processes and may be thought of as partitioned into n subsequences, one controlling each process. We do *not* assume that the request sequence is part of the schedule. The reason is that we want to be able to compare different algorithms on the same schedule even if these algorithms do not necessarily implement the same set of tasks.

Since in the present work we are only working with deterministic algorithms, we can assume that the request sequence and schedule are fixed in advance. This

assumption is not required for our results but allows the presentation to be simplified. A more general approach would be to assume that both are generated on the fly by an adaptive adversary.

We will also assume that the algorithms we consider are implementing *objects* which are abstract concurrent data structures with well-defined interfaces and correctness conditions. We will assume that these objects are manipulated by invoking *tasks* of some sort, that we can count the number of tasks completed by an algorithm implementing an object, and that we can distinguish correct implementations from incorrect implementations. Otherwise the details of objects will be left unspecified unless we are dealing with specific applications.

3 The Measures

Competitive throughput. The competitive throughput of an algorithm measures how many tasks an algorithm can complete in a given amount of time. We measure the algorithm against a champion algorithm that runs under the same schedule. We do not assume that both algorithms are given the same request sequence; we require only that the two request sequences be made up of tasks for the same object T.

Some notation: for each algorithm A, schedule σ, and request sequence R, define $\mathrm{done}(A, \sigma, R)$ to be the total number of tasks completed by all processes when running A according to the schedule σ and request sequence R. Define $\mathrm{opt}_T(\sigma)$ to be $\max_{A^*, R^*} \mathrm{done}(A^*, \sigma, R^*)$, where A^* ranges over all correct implementations of T and R^* ranges over all request sequences composed of T-tasks. (Thus $\mathrm{opt}_T(\sigma)$ represents the performance of the best correct algorithm running on the best-case input for the schedule σ.)

Definition 1 *Let A be an algorithm that implements an object T. Then A is* k-throughput-competitive *for T if there exists a constant c such that, for any schedule σ and request sequence R,*

$$\mathrm{done}(A, \sigma, R) + c \geq \frac{1}{k} \mathrm{opt}_T(\sigma). \tag{1}$$

This definition follows the usual definition of competitive ratio.

Relative competitiveness. The full power of the competitive throughput measure only becomes apparent when we consider modular algorithms. Under many circumstances, it will be possible to show that an algorithm that uses an undetermined subroutine is competitive relative to the object implemented by that subroutine, in the sense that plugging in any competitive algorithm for the subroutine gives a competitive version of the algorithm as a whole. This intuition gives rise to the definition of relative throughput-competitiveness given below.

As in the definition of throughput-competitiveness, we consider a situation in which A is an algorithm implementing some object T. Here, however, we assume that A depends on a (possibly unspecified) subroutine implementing a different object U. For any specific algorithm B that implements U, we will write $A \circ B$ for the composition of A with B, i.e., for that algorithm which is obtained by running B whenever A needs to carry out a U-task.[5]

Definition 2 *An algorithm A is* k-throughput-competitive for T relative to U *if there exists a constant c such that for any B that implements U, and any schedule σ and request sequence R for which the ratios are defined,*

$$\frac{\mathrm{done}(A \circ B, \sigma, R) + c}{\mathrm{done}(B, \sigma, R_A)} \geq \frac{1}{k} \cdot \frac{\mathrm{opt}_T(\sigma)}{\mathrm{opt}_U(\sigma)}, \tag{2}$$

where R_A is the request sequence corresponding to the subroutine calls in A when running according to R and σ.

As in the preceding definition, the additive constant c is included to avoid problems with granularity. The condition that the ratios are defined (which in essence is just a requirement that σ be long enough for B to complete at least one U-task) is needed for the same reason.

4 Composition of Competitive Algorithms

Theorem 3 (Composition) *Let A be an algorithm that is k-throughput-competitive for T relative to U, and assume there exists a constant c such that for all schedules σ,*

$$c\,\mathrm{opt}_U(\sigma) \geq \mathrm{opt}_T(\sigma)\ .^{6} \tag{3}$$

Let B be an l-throughput-competitive algorithm for U. Then $A \circ B$ is kl-throughput-competitive for T.

It is worth noting that the theorem can be applied more than once to obtain a kind of transitivity. If A implements T competitively relative to U, B implements U competitively relative to V, and C implements V competitively, then applying the theorem first to B and C, and then to A and $B \circ C$, shows that $A \circ B \circ C$ is competitive.

5 The Write-Collect Object

The write-collect object acts like a set of n single-writer n-reader atomic registers and provides two operations for manipulating these registers. A *collect* operation returns the values of all of the registers, with the guarantee that any value returned was not overwritten before the start of the collect. (This condition is trivially satisfied by an algorithm that reads all n registers directly,

[5] For this definition it is important that A not execute any operations that are not provided by U. In practice the difficulties this restriction might cause can often be avoided by treating U as a composite of several different objects.

[6] Intuitively, this condition requires that for every T-task that we can finish, we can finish at least $1/c$ U-tasks. In effect it says that T is not a weaker object than U is.

but may be more difficult for an algorithm in which processes may learn values indirectly from other processes.) A *write-collect* operation writes a new value to the process's register and then performs a collect. It must satisfy this rather weak serialization condition: given two write-collects a and b, if the first operation of a precedes the first operation of b, then b returns the value written by a as part of its vector; but if the first operation of a follows the last operation of b, then b does not return the value written by a. (Again, this condition is trivially satisfied by the naive algorithm that simply does a write followed by n reads.)

The write-collect operation is motivated by the fact that many shared-memory algorithms execute collects interspersed with write operations (some examples are given in Section 8.

6 A Throughput-Competitive Implementation of Write-Collect

To implement a write-collect we start with the cooperative collect algorithm of [3]. This algorithm has several desirable properties:

1. All communication is through a set of single-writer registers, one for each process, and the first operation of the cooperative collect is a write operation.
2. No collect operation ever requires more than $2n$ steps to complete.
3. For any schedule, and any set of collects that are in progress at some time t, there is a bound on the total number of steps required to complete these collects. (Showing this bound is nontrivial; a proof can be found in [3]).

These properties are what we need from a cooperative collect implementation to prove that it gives a throughput-competitive write-collect. The first property allows us to ignore the distinction between collect and write-collect operations (at least in the candidate): we can include the value written by the write-collect along with this initial write, and thus trivially extend a collect to a write-collect with no change in the behavior of the algorithm. In effect, our throughput-competitive write-collect algorithm is simply the latency-competitive collect of [3], augmented by merging the write in a write-collect with the first write done as part of the collect implementation.

The last two properties tell us that any write-collects in progress at a given time will finish soon, from which it would seem to follow that the number of write-collects that are completed will be large. However, to show that the number of collects and/or write-collects completed by a modified cooperative collect algorithm with these properties is proportional to the number completed by a champion algorithm is not easy. In Section 6.1, we show a bound on the throughput-competitiveness of any algorithm with the above properties.

6.1 Proof of Competitiveness

In order to show that a collect algorithm gives a throughput-competitive write-collect, we must do two things. First, we must prove a lower bound on the number of collects done by the algorithm based on some properties of the schedule. As noted above, for algorithms in which a collect starts with a write this gives a lower bound on the number of mixed collects and write-collects as well. Second, we must show that those same properties of the schedule imply an upper bound on the number of collects done by any correct algorithm. Since a write-collect includes a collect, an upper bound on the collects done by the champion implies a corresponding upper bound on the number of mixed collects and write-collects. Thus in both cases we can concentrate solely on collect operations.

Given an algorithm A and a schedule σ, define the *private latency* of a process p at time t to be the number of steps (i.e., atomic read and write operations) done by p after t and before the end of the last collect that p started at or before t. If this quantity is bounded for all σ, p, and t, denote the bound by $\mathrm{UPL}(A)$.

Similarly, given an algorithm A, the *collective latency* at time t is defined [3] as the sum over all processes p of the private latency for p at time t, and is denoted by $CL(A,t)$. If this quantity is bounded for all A and σ, denote the bound by $\mathrm{UCL}(A)$. Note that $\mathrm{UCL}(A)$ may be much smaller than $n \cdot \mathrm{UPL}(A)$ as concurrent processes may cooperate to finish their collects quickly; for example, in the cooperative collect protocol of [3], $\mathrm{UPL} = 2n$ but $\mathrm{UCL} = O(n^{3/2}\log^2 n)$.

We denote by $\mathrm{FCTh}_p(A,t)$ the *fractional collective throughput* in algorithm A of a process p at point t in time, and define it inductively as follows. When $t = 0$, $\mathrm{FCTh}_p(A,0) = 0$. If at time t, some process q (which may or may not be equal to p) performs a step as part of a collect operation C, then $\mathrm{FCTh}_p(A,t) = \mathrm{FCTh}_p(A,t-1) + \frac{1}{\mathrm{UCL}(A)+2n}$ if at least one of the following holds:

1. p is in the middle of a collect operation that started no earlier than C started;
2. This step of q is the last step it performs before p starts a new collect operation; or
3. This step of q is the first step it performs after the last collect completed by p.

If none of the conditions hold, then $\mathrm{FCTh}_p(A,t) = \mathrm{FCTh}_p(A,t-1)$.

A rough intuition is that the increment $\frac{1}{\mathrm{UCL}(A)+2n}$ represents how much of the current collective latency is "used up" by q. The $2n$ in the denominator is an artifact of the operations (up to two per process) in classes (2) or (3), and should not be confused with the value of $\mathrm{UPL}(A)$ for any particular algorithm.

Analogously, the *fractional private throughput* in algorithm A of a process p at point t in time is denoted by $FPTh_p(A,t)$ and is defined inductively as follows. When $t = 0$, $\mathrm{FTh}_p(A,0) = 0$. If at time t,

p performs a step as part of a collect operation, then $\mathrm{FPTh}_p(A,t) = \mathrm{FPTh}_p(A,t-1) + \frac{1}{\mathrm{UPL}(A)}$; otherwise, $\mathrm{FPTh}_p(A,t) = \mathrm{FPTh}_p(A,t-1)$. Again, the intuition here is that the increment $\frac{1}{\mathrm{UPL}(A)}$ represents how much of p's private latency is used up by the step at time t.

The *fractional throughput* in algorithm A of a process p at point t in time is denoted by $\mathrm{FTh}_p(A,t)$ and is defined as $(\mathrm{FCTh}_p(A,t) + \mathrm{FPTh}_p(A,t))/2$.

Lemma 4 *Let A be a cooperative collect algorithm for which $\mathrm{UPL}(A)$ and $\mathrm{UCL}(A)$ are defined. Then in any execution of A, the total number of collects completed by all processes by time t in A is at least $\sum_p \mathrm{FTh}_p(A,t) - n$.*

The n corresponds to collects that have not yet finished at time t.

Lemma 5 *Let A be an algorithm for which $\mathrm{UPL}(A)$ and $\mathrm{UCL}(A)$ are defined. Let $I = (t_1, t_2]$ be a time interval in which n steps are performed and m processes perform steps. Then $\sum_p \mathrm{FTh}_p(t_2) - \sum_p \mathrm{FTh}_p(t_1) \geq \frac{1}{2}\left(\frac{n}{\mathrm{UPL}} + \frac{m^2}{2\cdot(\mathrm{UCL}+2n)}\right)$.*

Lemma 6 *Let A be a cooperative collect algorithm. Let $I = (t_1, t_2]$ be a time interval in which n steps are performed and m processes perform steps. Then, the number of collects completed in this interval is at most m.*

Combining the results of Lemmas 4, 5 and 6 gives:

Theorem 7 *Let A be a cooperative collect algorithm for which $\mathrm{UPL}(A)$ and $\mathrm{UCL}(A)$ are defined. Then A is $\sqrt{\frac{8(\mathrm{UCL}+2n)\,\mathrm{UPL}}{n}}$-throughput-competitive.*

The theorem can be applied to give an immediate upper bound on the throughput-competitiveness of any cooperative collect algorithm A for which the private and collective latencies are always bounded. For example, plugging the bounds on the private and collective latencies of the faster algorithm of [3] into the formula in Theorem 7, it immediately follows that this algorithm is $O(n^{3/4}\log^2 n)$-throughput-competitive. Since that algorithm has the property that the first operation of any collect is a write, and the write-collect operation is strictly stronger than collect, this immediately gives us $O(n^{3/4}\log^2 n)$-throughput-competitive write-collect algorithm as well. This bound is the first to cross the trivial n bound, and, as implied by the lower bound of the following section, this bound is nearly optimal. (A similar discussion applies to the slower algorithm of [3].)

7 Lower Bound on Throughput Competitiveness of Collect

Theorem 8 *No cooperative collect protocol has a throughput competitiveness less than $\Omega(\sqrt{n})$.*

Sketch of Proof: To overcome the additive constant, we build up an arbitrarily-long schedule out of phases, where in each phase the ratio of champion to candidate collects is $\Omega(\sqrt{n})$. The essential idea is that in each phase, most of the work will be done by a single "patsy" process, chosen at the beginning of the phase. Profiting from the patsy's labors, in the champion algorithm, will be $\sqrt{n}$ "active" processes (fixed for all phases). These same processes, in the candidate algorithm, will not benefit from the patsy's work, since we will terminate a phase as soon as any active process discovers the patsy or completes a collect started in that phase.

Each phase consists of one or more rounds. The schedule for a round consists of one step for each active process, followed by $n+\sqrt{n}+1$ steps for the patsy, and ended by one additional step for each active processor. In the champion algorithm: (1) the active processes write out timestamps; (2) the patsy reads these timestamps; (3) the patsy reads the registers; (4) the patsy writes out a summary of the register values with the timestamps attached; and (5) the active processes read this summary. Because the patsy reads the timestamps before it reads the registers, the active processes will know that the patsy's values are fresh. Thus the champion completes $\sqrt{n}+1$ collects per round.

In the candidate, each active processor completes exactly one collect *per phase*. We enforce this by ending a phase as soon as some active processor obtains all the register values or finds the patsy; to avoid difficulties we attach a special "cleanup round" in which each processor is given just enough steps to complete its current collect. Since no active processor reads the patsy's register until the last round of the phase, the active processes cannot mark the values they write with a timestamp that will allow the patsy to trust them. Thus, the patsy will complete roughly the same one collect per round as in the champion.

The only issue left is how to choose a patsy that will not be found quickly. If the patsy is chosen uniformly at random, expected $\Omega(n)$ active-processor reads will be needed either to find it or to read all the register values, giving $\Omega(\sqrt{n})$ expected rounds. Ignoring some technical details, this gives roughly $\Omega(n)$ collects per phase for the champion, versus $O(\sqrt{n})$ for the candidate, for a ratio of $\Omega(\sqrt{n})$. ■

8 Applications

Armed with a throughput-competitive write-collect algorithm and Theorem 3, it is not hard to obtain throughput-competitive versions of many well-known shared-memory algorithms. Examples include snapshot algorithms [2, 5, 8, 11, 13, 23], the bounded round numbers abstraction [29], concurrent timestamping systems [25, 28, 32, 34, 35, 40], and time-lapse snapshot [28]. Here we elaborate on some simple examples.

Atomic snapshots. A *snapshot* object simulates an array of n single-writer registers that support a *scan-update* operation which writes a value to one of the registers (an "update") and returns a vector of values for

all of the registers (a "scan"). A scan-update is distinguished from the weaker write-collect operation by virtue of a much stronger serialization condition; informally, this says that the vector of scanned values must appear to be a picture of the registers at some particular instant during the execution. Snapshot objects are very useful tools for constructing more complicated shared-memory algorithms, and they have been extensively studied [2, 5, 8, 11, 23] culminating in the protocol of Attiya and Rachman [13] which uses only $O(\log n)$ alternating writes and collects to complete a scan-update operation.

We will apply Theorem 3 to get a competitive snapshot. Let T be a snapshot object and U a write-collect object. Because a scan-update can be used to simulate a write-collect or collect, we have $\mathrm{opt}_T(\sigma) \leq \mathrm{opt}_U(\sigma)$ for any schedule σ, and the inequality (3) is satisfied. Furthermore, if A is the Attiya-Rachman snapshot, B is any write-collect algorithm, and R is any request sequence of scan-update operations, then for any σ,

$$\frac{\mathrm{done}(A \circ B, \sigma, R) + 1}{\mathrm{done}(B, \sigma, R_A)} \geq \frac{1}{O(\log n)} \geq \frac{1}{O(\log n)} \cdot \frac{\mathrm{opt}_T(\sigma)}{\mathrm{opt}_U(\sigma)}$$

which means that the Attiya-Rachman snapshot is $O(\log n)$-throughput-competitive relative to write-collect. By Theorem 3, plugging in a k-throughput-competitive implementation of write-collect thus gives an $O(k \log n)$-throughput-competitive snapshot protocol.

Bounded round numbers. A large class of wait-free algorithms that communicate via single-writer multi-reader atomic registers have a communication structure based on *asynchronous rounds*. Starting from round 1, at each round the process performs a computation, and then advances its round number and proceeds to the next round. A process's actions do not depend on its exact round number but only on the distance of its current round number from those of other processes. Moreover, the process's actions are not affected by any process whose round number lags behind its own by more than a finite limit. The round numbers increase unboundedly over the lifetime of the system.

Dwork, Herlihy and Waarts [29] introduced the *bounded round numbers* abstraction, which can be plugged into any algorithm that uses round numbers in this fashion, transforming it into a bounded algorithm. The bounded round numbers implementation in [29] provides four operations of varying difficulty; however, the use of these operations is restricted. As a result, we can coalesce these operations into a single operation, an *advance-collect*, which advances the current process's round number to the next round and collects the round numbers of the other processes. Using their implementation, only $O(1)$ alternating writes and collects are needed to implement an advance-collect.

Again we can apply Theorem 3. Let T be a bounded round numbers object (i.e., an object providing the advance-collect operation) and let U be write-collect. Because an advance-collect must gather information from every process in the system, it implicitly contains a collect, and thus $\mathrm{opt}_T(\sigma) \leq \mathrm{opt}_U(\sigma)$ for all schedules σ. An argument similar to that used above for the Attiya-Rachman snapshot thus shows that plugging a k-throughput-competitive implementation of write-collect into the Dwork-Herlihy-Waarts bounded round numbers algorithm gives an $O(k)$-throughput-competitive algorithm.

The bounded round numbers object provides an interesting example of how composite objects can be built. Suppose that we have an algorithm that uses both the advance-collect operation from a bounded round numbers object to organize its computation, and the write-collect and collect operations provided by a write-collect object to communicate between processes. We can consider all three operations as being provided by a composite object obtained from a bounded round numbers object built on top of write-collect object, which passes requests for collects and write-collects directly to the underlying write-collect object. It is not hard to see that this composite object will be $O(1)$-competitive relative to write-collect, and that it will thus be $O(k)$-throughput-competitive if an $O(k)$-throughput-competitive write-collect implementation is used.

Given an algorithm that implements a third object whose operations are at least as difficult as a collect (the cheapest of the three operations of the composite object), and that uses at most l advance-collects, collects, and write-collects for each operation, Theorem 3 applies to show that this algorithm will be $O(kl)$-throughput-competitive.

9 Acknowledgments

We are indebted to Miki Ajtai and Cynthia Dwork for very helpful discussions, and to Maurice Herlihy on helpful comments on the presentation of this work. We also thank Amos Fiat for his encouragement.

References

[1] K. Abrahamson. On achieving consensus using a shared memory. In *Proc. 7th ACM Symposium on Principles of Distributed Computing*, pp. 291–302, August 1988.

[2] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit. Atomic Snapshots of Shared Memory. In *Journal of the ACM*, Vol. 40, No. 4, pages 872-890, September 1993. Preliminary version appears in *Proc. 9th ACM Symposium on Principles of Distributed Computing*, pp. 1-13, 1990.

[3] M. Ajtai, J. Aspnes, C. Dwork, and O. Waarts. A theory of competitive analysis for distributed algorithms. In *Proc. 33rd IEEE Symposium on Foundations of Computer Science*, pp. 401–411, November 1994. Full version available.

[4] N. Alon, G. Kalai, M. Ricklin, and L. Stockmeyer. Lower bounds on the competitive ratio for mobile user tracking and distributed job scheduling. In *Proc. 33rd IEEE Symposium on Foundations of Computer Science*, pages 334-343, October 1992.

[5] J. Anderson. Composite Registers. In *Distributed Computing*, Vol. 6, No. 3, pages 141-154. Preliminary version appears in *Proc. 9th ACM Symposium on Principles of Distributed Computing*, pp. 15-30, August 1990.

[6] J. Aspnes. Time- and space-efficient randomized consensus. *Journal of Algorithms* 14(3):414–431, May 1993. An earlier version appeared in *Proc. 9th ACM Symposium on Principles of Distributed Computing*, pp. 325–331, August 1990.

[7] J. Aspnes and M.P. Herlihy. Fast randomized consensus using shared memory. In *Journal of Algorithms 11*(3), pp.441–461, September 1990.

[8] J. Aspnes and M.P. Herlihy. Wait-free data structures in the Asynchronous PRAM Model. In *Proceedings of the 2nd Annual Symposium on Parallel Algorithms and Architectures*, July 1990, pp. 340–349, Crete, Greece.

[9] J. Aspnes and O. Waarts. Randomized consensus in expected $O(n \log^2 n)$ operations per processor. In *Proc. 33rd IEEE Symposium on Foundations of Computer Science*, pp. 137–146, October 1992. To appear, *SIAM Journal on Computing*.

[10] J. Aspnes and O. Waarts. Modular competitiveness for distributed algorithms. Manuscript 1995.

[11] H. Attiya, M.P. Herlihy, and O. Rachman. Efficient atomic snapshots using lattice agreement. Technical report, Technion, Haifa, Israel, 1992. A preliminary version appeared in proceedings of *the 6th International Workshop on Distributed Algorithms*, Haifa, Israel, November 1992, (A. Segall and S. Zaks, eds.), Lecture Notes in Computer Science #647, Springer-Verlag, pp. 35–53.

[12] H. Attiya, A. Herzberg, and S. Rajsbaum. Optimal clock synchronization under different delay assumptions. In *Proc. 12th ACM Symposium on Principles of Distributed Computing*, pp. 109–120, Aug. 1993.

[13] H. Attiya and O. Rachman. Atomic snapshots in $O(n \log n)$ operations. In *Proc. 12th ACM Symposium on Principles of Distributed Computing*, pp. 29–40, Aug. 1993.

[14] B. Awerbuch and Y. Azar. Local optimization of global objectives: Competitive distributed deadlock resolution and resource allocation. In *Proc. 35th IEEE Symposium on Foundations of Computer Science*, pp. 240–249, November 1994.

[15] B. Awerbuch, Y. Bartal, and A. Fiat. Competitive distributed file allocation. In *Proc. 25th ACM Symposium on Theory of Computing*, pp. 164–173, May 1993.

[16] B. Awerbuch, Y. Bartal, and A. Fiat. Heat and Dump: competitive distributed paging. In *Proc. 34th IEEE Symposium on Foundations of Computer Science*, pp. 22–31, November 1993.

[17] B. Awerbuch, S. Kutten, and D. Peleg. Competitive distributed job scheduling. In *Proc. 24th ACM Symposium on Theory of Computing*, pp. 571–580, May 1992.

[18] B. Awerbuch and D. Peleg. Sparse partitions. In *Proc. 31st IEEE Symposium on Foundations of Computer Science*, pp. 503–513, November 1990.

[19] Y. Bartal, A. Fiat, and Y. Rabani. Competitive algorithms for distributed data management. In *Proc. 24th ACM Symposium on Theory of Computing*, pp. 39–50, 1992.

[20] E. Borowsky and E. Gafni. Immediate atomic snapshots and fast renaming. In *Proc. 12th ACM Symposium on Principles of Distributed Computing*, pp. 41–51, August 1993.

[21] Y. Bartal and A. Rosen. The distributed k-server problem – A competitive distributed translator for k-server algorithms. In *Proc. 33rd IEEE Symposium on Foundations of Computer Science*, pp. 344–353, October 1992.

[22] G. Bracha and O. Rachman. Randomized consensus in expected $O(n^2 \log n)$ operations. *Proceedings of the Fifth International Workshop on Distributed Algorithms*. Springer-Verlag, 1991.

[23] T. Chandra and C. Dwork. Using consensus to solve atomic snapshots. *Submitted for Publication*

[24] B. Chor, A. Israeli, and M. Li. Wait-Free Consensus Using Asynchronous Hardware. In *SIAM Journal on Computing*, Vol. 23, No. 4, pages 701-712, August 1994. Preliminary version appears in *Proc. 6th ACM Symposium on Principles of Distributed Computing*, pages 86-97, 1987.

[25] D. Dolev and N. Shavit. Bounded Concurrent Time-Stamp Systems are Constructible! To appear in *SIAM Journal on Computing*. Preliminary version appears in *Proc. 21st ACM Symposium on Theory of Computing*, pages 454-465, 1989. An extended version appears in IBM Research Report RJ 6785, March 1990.

[26] D. Dolev, R. Reischuk, and H.R. Strong. Early stopping in Byzantine agreement. In *Journal of the ACM*, 34:7, Oct. 1990, pp. 720–741. First appeared in: Eventual is Earlier than Immediate, IBM RJ 3915, 1983.

[27] C. Dwork, J. Halpern, and O. Waarts. Accomplishing work in the presence of failures. In *Proc. 11th ACM Symposium on Principles of Distributed Computing*, pp. 91–102, 1992.

[28] C. Dwork, M.P. Herlihy, S. Plotkin, and O. Waarts. Time-lapse snapshots. *Proceedings of Israel Symposium on the Theory of Computing and Systems*, 1992.

[29] C. Dwork, M.P. Herlihy, and O. Waarts. Bounded round numbers. In *Proc. 12th ACM Symposium on Principles of Distributed Computing*, pp. 53–64, 1993.

[30] C. Dwork, M.P. Herlihy, and O. Waarts. Contention in shared memory algorithms. To appear in the *Journal of the ACM*. Preliminary version appears in *Proc. 25th ACM Symposium on Theory of Computing*, pages 174-183, May 1993. Expanded version: Digital Equipment Corporation Technical Report CRL 93/12.

[31] C. Dwork and Y. Moses. Knowledge and common knowledge in a Byzantine environment: crash failures. In *Information and Computation 88*(2) (1990), originally in *Proc. TARK* 1986.

[32] C. Dwork and O. Waarts. Simple and efficient bounded concurrent timestamping or bounded concurrent timestamp systems are comprehensible! To appear in the *Journal of the ACM*. Preliminary version appears in *Proc. 24th ACM Symposium on Theory of Computing*, pp. 655–666, 1992.

[33] M. Fischer and A. Michael, Sacrificing serializability to attain high availability of data in an unreliable network. Research Report 221, Yale U., Feb. 1982.

[34] R. Gawlick, N. Lynch, and N. Shavit. Concurrent timestamping made simple. *Proceedings of Israel Symposium on Theory of Computing and Systems*, 1992.

[35] S. Haldar. Efficient bounded timestamping using traceable use abstraction - Is writer's guessing better than reader's telling? Technical Report RUU-CS-93-28, Department of Computer Science, Utrecht, September 1993.

[36] J. Y. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment, *Journal of the Association for Computing Machinery*, Vol 37, No 3, January 1990, pp. 549–587. A preliminary version appeared in *Proc. 3rd ACM Symposium on Principles of Distributed Computing*, 1984.

[37] J.Y. Halpern, Y. Moses, and O. Waarts. A characterization of eventual Byzantine agreement. In *Proc. 9th ACM Symposium on Principles of Distributed Computing*, pp. 333–346, August 1990.

[38] M.P. Herlihy. Randomized wait-free concurrent objects. In *Proc. 10th ACM Symposium on Principles of Distributed Computing*, August 1991.

[39] A. Israeli and M. Li. Bounded Time Stamps. In *Distributed Computing*, Vol. 6, No. 4, pages 205-209. Preliminary version appears in *Proc. 28th IEEE Symposium on Foundations of Computer Science*, pp. 371-382, October 1987.

[40] A. Israeli and M. Pinhasov. A Concurrent Time-Stamp Scheme which is Linear in Time and Space. In *Proceedings of the 6th International Workshop on Distributed Algorithms, LNCS 647*, Springer-Verlag, pages 95-109, 1992.

[41] E. Koutsoupias and C.H. Papadimitriou. Beyond competitive analysis. In *Proc. 33rd IEEE Symposium on Foundations of Computer Science*, pp. 394–400, November 1994.

[42] L. M. Kirousis, P. Spirakis and P. Tsigas. Reading many variables in one atomic operation: solutions with linear or sublinear complexity. In *Proceedings of the 5th International Workshop on Distributed Algorithms*, 1991.

[43] Y. Moses and M.R. Tuttle. Programming simultaneous actions using common knowledge. *Algorithmica 3*(1), pp. 121–169, 1988 Preliminary version appeared in *Proc. 28th IEEE Symposium on Foundations of Computer Science*, pp. 208–221, 1986.

[44] G. Neiger. Using knowledge to achieve consistent coordination in distributed systems. Manuscript, 1990.

[45] G. Neiger and M. R. Tuttle. Common knowledge and consistent simultaneous coordination. In *Proceedings of the 4th International Workshop on Distributed Algorithms*, 1990.

[46] C.H. Papadimitriou and M. Yannakakis. Linear programming without the matrix. In *Proc. 25th ACM Symposium on Theory of Computing*, pp. 121–129, May 1993.

[47] B. Patt-Shamir and S. Rajsbaum. A theory of clock synchronization. In *Proc. 26th ACM Symposium on Theory of Computing*, pp. 810–819, May 1994.

[48] Y. Riany, N. Shavit and D. Touitou. Practical Snapshots. In *Proceedings of Israel Symposium on Theory of Computing and Systems*, 1994.

[49] M. Saks, N. Shavit, and H. Woll. Optimal time randomized consensus — making resilient algorithms fast in practice. In *Proceedings of the 2nd ACM-SIAM Symposium on Discrete Algorithms*, pp. 351–362, 1991.

[50] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Comm. of the ACM 28*(2), pp. 202–208, 1985.

[51] P.M.B. Vitányi and B. Awerbuch. Atomic Shared Register Access by Asynchronous Hardware. In *Proc. 27th IEEE Symposium on Foundations of Computer Science*, pp. 233-243, 1986. Errata in *Proc. 28th IEEE Symposium on Foundations of Computer Science*, 1987.