Compositional Competitiveness for Distributed Algorithms*

James Aspnes† Orli Waarts‡

June 14, 2004

Abstract

We define a 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. This new measure complements the *latency* measure of Ajtai *et al.* [3], which measures how quickly an algorithm can finish tasks that start at specified times. The novel feature of the throughput measure, which distinguishes it from the latency measure, is that it is compositional: it supports a notion of algorithms that are competitive *relative to* a class of subroutines, with the property that an algorithm that is k-competitive relative to a class of subroutines, combined with an ℓ-competitive member of that class, gives a combined algorithm that is $k\ell$-competitive.

In particular, we prove the throughput-competitiveness of a class of algorithms for *collect operations*, in which each of a group of n processes obtains all values stored in an array of n registers. Collects are a fundamental building block of a wide variety of shared-memory distributed algorithms, and we show that several such algorithms are competitive relative to collects. Inserting a competitive collect in these algorithms gives the first examples of competitive distributed algorithms obtained by composition using a general construction.

*An earlier version of this work appeared as "Modular competitiveness for distributed algorithms," in *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing*, pages 237–246, Philadelphia, Pennsylvania, 22–24 May 1996.

†Yale University, Department of Computer Science, 51 Prospect Street/P.O. Box 208285, New Haven CT 06520-8285. Supported by NSF grants CCR-9410228, CCR-9415410, CCR-9820888, and CCR-0098078. 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

The tool of competitive analysis was proposed by Sleator and Tarjan [49] 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 on-line 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.

We would like to apply competitive analysis to the design of asynchronous distributed algorithms, where the input sequence may reflect both user commands and the timing of events in the underlying system. Our goal, following previous work of Ajtai *et al.* [3], is to find competitive algorithms for core problems in distributed computing that can then be used to speed up algorithms for solving more complex problems. To this end, we need a definition of competitive performance that permits *composition*: the construction of a competitive algorithm by combining a competitive superstructure with a competitive subroutine. Our efforts to find such a definition have led us to a notion of *competitive throughput*, which counts the number of operations or tasks that can be completed by some algorithm in a fixed time frame. We show that for a particular problem, the *cooperative collect problem*, it is possible both (a) to obtain cooperative collect algorithms with good competitive throughput and (b) to use these algorithms as subroutines in many standard algorithms not specifically designed for competitive analysis, and thereby obtain competitive versions of these algorithms.

We begin with a short history of competitive analysis in distributed algorithms (Section 1.1), followed by a discussion of the cooperative collect primitive (Section 1.2) and an overview of our approach and the organization of the rest of the paper (Section 1.3).

1.1 Competitive analysis and distributed algorithms

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.

Early work on distributed job scheduling and data management [4, 16, 17, 19–22] took the approach of comparing a distributed on-line algorithm to a global-control off-line algorithm. However, as noted in [17], using a global-control algorithm as a reference has the unfortunate side-effect of forcing the on-line algorithm to compete not only against algorithms that can predict the future but against algorithms in which each process can deduce what all other processes are doing at no cost.

While such a measure can be useful for algorithms that are primarily concerned with managing resources, it unfairly penalizes algorithms whose main purpose is propagating information. Ajtai *et al.* [3] described a more refined approach in which a *candidate* distributed algorithm is compared to an optimal *champion*. In their *competitive latency* model, both the candidate and champion are distributed algorithms. Both are subject to an unpredictable *schedule* of events in the system and both must satisfy the same correctness condition for all possible schedules. The difference is that the adversary may supply a different champion optimized for each individual schedule when measuring performance.

1.2 Cooperative collect

The competitive latency model was initially designed to analyze a particular problem in distributed computing called *cooperative collect*, first abstracted by Saks *et al.* [48]. The cooperative collect problem arises in asynchronous shared-memory systems built from single-writer registers.[1] In order to observe the state of the system, a process must read $n-1$ registers, one for each of the other processes. The simplest implementation of this operation is to have the process carry out these $n-1$ reads by itself. However, if many processes are trying to read the same set of registers simultaneously, some

[1] A single-writer register is one that is "owned" by some process and can only be written to by its owner.

of the work may be shared between them.

A *collect operation* is any procedure by which a process may obtain the values of a set of n registers (including its own). The correctness conditions for a collect are those that follow naturally from the trivial implementation consisting of $n-1$ reads. The process must obtain values for each register, and those values must be *fresh*, meaning that they were present in the register at some time between when the collect started and the collect finished.

Curiously, the trivial implementation is the one used in almost all of the many asynchronous shared-memory algorithms based on collects, including algorithms for consensus, snapshots, coin flipping, bounded round numbers, timestamps, and multi-writer registers [1, 2, 5, 6, 8, 9, 11, 12, 14, 23–25, 27–29, 31, 33, 34, 36, 38–40, 50]. (Noteworthy exceptions are [47, 48], which present interesting collect algorithms that do not follow the pattern of the trivial algorithm, but which depend on making strong assumptions about the schedule.) Part of the reason for the popularity of this approach may be that the trivial algorithm is optimal in the worst case: a process running in isolation has no alternative but to gather all $n-1$ register values itself.

Ajtai *et al.*'s [3] hope was that a cooperative collect subroutine with a good competitive ratio would make any algorithm that used it run faster, at least in situations where the competitive ratio implies that the subroutine outperforms the trivial algorithm. To this end, they constructed the first known competitive algorithm for cooperative collect and showed a bound on its competitive latency. Unfortunately, there are technical obstacles in the competitive latency model that make it impossible to *prove* that an algorithm that uses a competitive collect is itself competitive. The main problem is that the competitive latency includes too much information in the schedule: in addition to controlling the timing of events in the underlying system such as when register operations complete, it specifies when high-level operations such as collects begin. So the competitive latency model can only compare a high-level algorithm to other high-level algorithms that use collects at exactly the same times and in exactly the same ways.

1.3 Our approach

In the present work, we address this difficulty by replacing the competitive latency measure with a *competitive throughput* measure that assumes that the candidate and champion face the same behavior in the system, but breaks the connection between the tasks carried out by the candidate and champion algorithms. This model is described in detail in Section 3. The

intuition 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). Previous work that compares a distributed algorithm with a global control algorithm [4, 16, 17, 19–22] 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. In effect, these models use a competitive input but a worst-case schedule. The competitive latency model of [3] applies the same input and schedule to both the on-line and the off-line algorithms. In contrast, we assume a worst-case *input* but a competitive *schedule*. Assuming a worst-case input means an algorithm must perform well in any context— including as part of a larger algorithm. At the same time, comparing the algorithm to others with the same schedule gives a more refined measure of the algorithm's response to bad system behavior than a pure worst-case approach.

The competitive throughput model solves the problem of comparing an algorithm A using a competitive subroutine B against an algorithm A^* that uses a subroutine B^* implementing the same underlying task, but it does not say anything about what happens when comparing A to an optimal A^* that does not call B^*. For this we need an additional tool that we call *relative competitiveness*, described in Section 4. We show (Theorem 4) that if an algorithm A is k-relative-competitive with respect to an underlying subroutine B, and B is itself l-competitive, then the combined algorithm $A \circ B$ is kl-competitive, even against optimal algorithms A^* that do not use B.

To demonstrate the applicability of these techniques, we show in Section 5 that the results of [3] can be extended to bound the competitive throughput of their algorithm; in fact, our techniques apply to any algorithm for which we have a bound on an underlying quantity that [3] called the *collective latency*, a measure of the total work needed to finish all tasks in progress at any given time. (This result has been used since the conference appearance of the present work by Aspnes and Hurwood [10] to prove low competitive throughput for an algorithm that improves on the algorithm of [3].) We show in Section 6 that relative competitiveness, combined with a throughput-competitive collect algorithm, does in fact give throughput-competitive solutions to problems such as atomic snapshot [2, 5, 9, 12, 14] and bounded round numbers [29]. We argue that most algorithms that use collects can be shown to be throughput-competitive using similar techniques.

Finally, in Section 7 we discuss some related approaches to analyzing distributed algorithms and consider what questions remain open.

2 Model

We use as our underlying model the wait-free shared-memory model of [37], using atomic single-writer multi-reader registers as the means of communication between processes. Because the registers are atomic, we can represent an execution as an interleaved sequence of steps, each of which is a read or write of some register. 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 and the contents of the registers). The adversary decides at each time unit which process gets to take the next step; these decisions are summarized in a *schedule*, which formally is just a sequence of process id's.

The algorithms we consider implement *objects*, which are abstract concurrent data structures with well-defined interfaces and correctness conditions. We assume that:

1. The objects are manipulated by invoking *tasks* of some sort;

2. That each instance of a task has a well-defined *initial operation* and a well-defined *final operation* (which may equal the initial operation for simple tasks).

3. That the definition of an initial or final operation depends only on the operation and the preceding parts of the execution, so that the completion of a task is recognizable at the particular step of the execution in which its final operation is executed, without needing to observe any subsequent part of the execution, and so that the number of tasks completed in an execution can be defined simply by counting the number of final operations executed; and

4. That there is a predicate on object schedules that distinguishes correct executions from incorrect executions, so that correct implementations are defined as those whose executions always satisfy this correctness predicate.

Beyond these minimal assumptions, the details of objects will be left unspecified unless we are dealing with specific applications.

Each process has as input a *request sequence* specifying what tasks it must carry out. The request sequences are supplied by the adversary and are *not* part of the schedule, as we may wish to consider the effect of different request sequences while keeping the same schedule. We assume that the

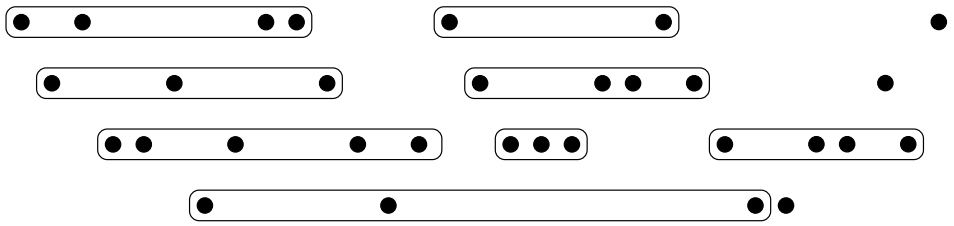

Figure 1: Throughput model. High-level operations (ovals) are implemented as a sequence of low-level steps (circles), which take place at times determined by the adversary. New high-level operations start as soon as previous operations end. Payoff to the algorithm is number of high-level operations completed.

request sequences are long enough that a process never runs out of tasks to perform.

The performance of an algorithm is measured by its competitive throughput, defined in Section 3. We contrast this definition with the competitive latency measure of Ajtai *et al.* [3] in Section 3.1. Building throughput-competitive algorithms by composition is described in Section 4.

3 Competitive throughput

The competitive throughput of an algorithm measures how many tasks an algorithm can complete with a given schedule. The assumption is that each process starts a new task as soon as each previous task is finished, as shown in Figure 1.

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 sequences; we only require that the two sets of request sequences be made up of tasks for the same object T. This assumption may seem unfair to the candidate algorithm, but it is necessary to allow algorithms to be composed. In reasoning about competitiveness compositionally, we compare the efficiency of a candidate B used as a subroutine in some higher-level algorithm A with the champion B^* used as a subroutine in some optimal higher-level algorithm A^*. In general we do not expect A and A^* to generate the same request sequences to B and B^* (hence the split between worst-case request sequences for B and best-case for B^*), but we can insist that both B and B^* run under the same schedule.

We start by introducing some notation. For each algorithm A, sched-

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

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

This definition follows the usual definition of competitive ratio [49]. The ratio is inverted since $\text{done}(A, \sigma, R)$ measures a payoff (the number of completed tasks) to be maximized instead of a cost to be minimized. The constant c is included to avoid problems that would otherwise arise from the granularity of tasks. On very short schedules, it might be impossible for A to complete even a single task, even though the best A^* could. Allowing the constant (which has minimal effect on longer schedules) gives a measure that more realistically describes the performance of A on typical schedules.

3.1 Comparison with competitive latency

The competitive throughput measure was inspired by the similar competitive latency measure of Ajtai *et al.* [3]. Competitive latency is not used in this paper, but we will give the definition from [3] to permit direct comparison between the two measures.

In the competitive latency model, the request sequences, including the times at which tasks start, are included in the schedule (see Figure 2). Thus the schedule σ includes both user input (the request sequences) and system timing (when each process is allowed to take a step). It is assumed that each task runs to completion, and that the process executing the task becomes idle until its next task starts; if the schedule calls for the process to carry out a step in between tasks, it performs a noop. The *total work* done by an algorithm A given a schedule σ, written $\text{work}(A, \sigma)$, is defined as the number of operations performed by processes outside of their idle periods.

The *competitive latency* of a candidate algorithm A is defined as [3]:

$$\sup_{\sigma} \frac{\text{work}(A, \sigma)}{\inf_{A^*} \text{work}(A^*, \sigma)},$$

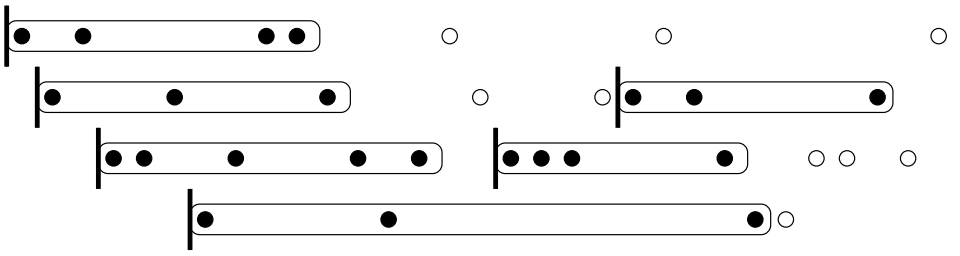

Figure 2: Latency model. New tasks (ovals) start at times specified by the schedule (vertical bars). Schedule also specifies timing of low-level steps (small circles). Cost to algorithm is number of low-level operations actually performed (filled circles), ignoring time steps allocated to processes in between tasks (empty circles).

where σ ranges over all schedules in which every task demanded of A is given enough steps to finish before a new task starts. We can rewrite this in a form closer to that of Definition 1, by writing that an algorithm A is k-latency-competitive if, for all schedules σ that permit A to finish its tasks, and all champion algorithms A^*,

$$\text{work}(A, \sigma) \leq k \, \text{work}(A^*, \sigma).$$

Note that this definition does not include an additive constant. The definition of competitive throughput does, to avoid problems with very short schedules in which the candidate A cannot complete any tasks. This problem does not arise with the competitive latency definition because of the restriction to schedules in which A can complete all assigned tasks.

Competitive latency has some advantages over competitive throughput. Because the request sequences are part of the schedule, it can be used to evaluate algorithms for which some tasks are much more expensive than others. In such a situation, the candidate in the competitive throughput model may be stuck with hard tasks while the champion breezes through easy ones. Competitive latency may thus be a better model than competitive throughput for measuring the performance of algorithms in isolation, though competitive throughput is a better measure for subroutines, as it allows composition using the results in Section 4. Often, difficulties with varying costs can also be ameliorated by joining cheap tasks to subsequent expensive ones, as is done in Sections 5.1 and 6.1.

On the other hand, competitive throughput removes some awkward features of competitive latency. In particular, the assumption that processes

are idle in between tasks (which is a necessary side-effect of specifying when in the sequence of operations each new task starts) may not be a good representation of how distributed algorithms are implemented in practice. A further concern is that if a process becomes idle quickly (say, during a brief lenient period in the schedule that allows quick termination), it is unaffected by harsher conditions that may arise later. This means in particular that a candidate algorithm that is only slightly slower than the champion it is competing against may find itself operating under much worse conditions, as the schedule suddenly gets worse as soon as the champion finishes.

The contrast between competitive latency and competitive throughput suggests a trade-off between competing notions of fair competition. Competitive latency treats different algorithms unfairly with respect to the schedule, by forcing a candidate to continue running in bad conditions after the champion has finished in good ones. But competitive throughput treats different algorithms unfairly with respect to the request sequences, because the definition explicitly assumes that the requests given to the candidate and champion may be different. It is not clear whether a more sophisticated definition could avoid both extremes, and produce a more accurate measure of the performance of a distributed algorithm compared to others running under similar conditions.

4 Composition of competitive algorithms

The full power of the competitive throughput measure only becomes apparent when we consider competitive algorithms built from competitive subroutines. In traditional worst-case analysis, an algorithm that invokes a subroutine k times at a cost of at most l time units each pays a total of kl time units. In a competitive framework, both the number of times the subroutine is called and the cost of each call to the subroutine may depend on system nondeterminism. The analogous quantity to the cost l of each subroutine call is the competitive ratio of the subroutine. What is an appropriate analog of the number of times k that the subroutine is called?

In Section 4.1, we define a notion of *relative competitiveness* that characterizes how well an algorithm uses a competitive subroutine. As shown in Section 4.2, algorithms that are k-throughput-competitive relative to an l-throughput-competitive subroutine are themselves throughput-competitive, with ratio kl. The definition of relative competitiveness (Definition 2) and the composition theorem that uses it (Theorem 4) yield a method for constructing competitive algorithms compositionally. Some examples of appli-

cations of this method appear in Section 6. To our knowledge this is the first example of a general composition theorem for competitive algorithms, even outside of a distributed setting.

4.1 Relative competitiveness

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.[2]

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,*

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

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.

The condition that the ratios are defined in (2) does create a curious loophole in the definition of relative competitiveness: if A implements some object T using an object U whose tasks can never be completed by a correct implementation, then the denominators in the inequality (2) are always zero, and thus A is vacuously zero-competitive relative to U. Similarly, an implementation B of U that never completes any tasks will be vacuously zero-competitive for U. Since $A \circ B$ is unlikely to be zero-competitive for T, to apply relative competitiveness we will need to exclude such pathologies. We do so using the following definition of *relative feasibility* of objects:

[2]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.

Definition 3 *Let T and U be objects. Say that T is* feasible relative to U *if there exists a constant c such that for all schedules σ,*

$$\mathrm{opt}_T(\sigma) \leq c \cdot \mathrm{opt}_U(\sigma) \tag{3}$$

In particular, if Definition 3 holds, then in any schedule where A completes at least one operation, opt_U is not zero and the right-hand side of (2) is defined. Furthermore, if B is competitive relative to U, then $\mathrm{done}(B, \sigma, R_A)$ is also nonzero for sufficiently long schedules, and the left-hand side of (2) is also defined.

4.2 The composition theorem

Theorem 4 describes under what conditions a relative-competitive algorithm combines with a competitive subroutine to yield a competitive algorithm.

Theorem 4 *Let A be an algorithm that is k-throughput-competitive for T relative to U, where T is feasible relative to U. Let B be an l-throughput-competitive algorithm for U. Then $A \circ B$ is kl-throughput-competitive for T.*

Proof: For the most part the proof requires only very simple algebraic manipulation of the definitions, but we must be careful about the constants and avoiding division by zero.

Fix σ and R. We can rewrite the inequality (2) as

$$(\mathrm{done}(A, \sigma, R) + c_A)\,\mathrm{opt}_U(\sigma) \geq \frac{1}{k}\,\mathrm{opt}_T(\sigma)\,\mathrm{done}(B, \sigma, R_A), \tag{4}$$

where c_A is the constant from the definition of relative competitiveness for A. Note that (4) holds even if one or both of the ratios in (2) is undefined, since in that case $\mathrm{done}(B, \sigma, R_A)$ must be zero and all other quantities are non-negative.

We can similarly rewrite (1) as

$$\mathrm{done}(B, \sigma, R_A) \geq \frac{1}{l}\,\mathrm{opt}_U(\sigma) - c_B \tag{5}$$

where c_B is a constant independent of σ and R. Plugging (5) into the right-hand side of (4) gives

$$(\mathrm{done}(A, \sigma, R) + c_A)\,\mathrm{opt}_U(\sigma) \geq \frac{1}{kl}\,\mathrm{opt}_T(\sigma)\,\mathrm{opt}_U(\sigma) - \frac{c_B}{k}\,\mathrm{opt}_T(\sigma),$$

which combines with the relative feasibility condition $\text{opt}_T(\sigma) \leq c_T \, \text{opt}_U(\sigma)$ to give

$$(\text{done}(A, \sigma, R) + c_A) \, \text{opt}_U(\sigma) \geq \frac{1}{kl} \, \text{opt}_T(\sigma) \, \text{opt}_U(\sigma) - \frac{c_B c_T}{k} \, \text{opt}_U(\sigma).$$

This last inequality gives the desired result, as either $\text{opt}_U(\sigma) > 0$ and we can divide out $\text{opt}_U(\sigma)$, or $\text{opt}_U(\sigma) = 0$ and thus $\text{opt}_T(\sigma) = 0$. In either case we have

$$\text{done}(A, \sigma, R) + c_A + \frac{c_B c_T}{k} \geq \frac{1}{kl} \, \text{opt}_T(\sigma). \tag{6}$$

■

Since we have dropped no terms in this derivation, if each of the inequalities (1), (2), and (3) used in the proof is tight, then (6) is also tight; so the additive constant $c_A + \frac{c_B c_T}{k}$ is the best possible that can be obtained without using additional information.

5 Cooperative collects

In this section, we define the *write-collect object*, which encapsulates the cooperative collect problem, and show how any cooperative collect algorithm satisfying certain natural criteria is throughput-competitive.

5.1 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.

1. 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.

2. A *write-collect* operation writes a new value to the process's register and then performs a collect.

The write-collect operation must satisfy a 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.

- If the first operation of a follows the last operation of b, then b does not return the value written by a.
- If the first operation of a occurs during the execution of b, b may return either the value written by a, or the previous value in the register written to by a.

A trivial implementation of write-collect might consist of a write followed immediately by n reads.

Our definition of 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 6). Treating write and collect as separate operations, though in many ways a more natural approach, also leads to difficulties in applying competitive throughput, as a candidate doing only expensive collects might find itself in competition with a champion doing only cheap writes.

5.2 Competitive algorithms for write-collect

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

1. All communication is through a set of single-writer registers, one for each process, and the first step of each collect operation is a write.
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 of $O(n^{3/2} \log^2 n)$ on the total number of steps required to complete these collects.

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 give two complementary bounds on the number of steps needed to finish collects in progress at any given time. The bound on

the total work to finish a set of simultaneous collects, called the *collective latency*, shows that processes can combine their efforts effectively when many are running simultaneously. The bound on the work done by any individual process, called the *private latency*, applies when only a few processes are running. We show, in Section 5.3, that any algorithm A with collective latency $\mathrm{CL}(A)$ and private latency $O(n)$ is $O\left(\sqrt{CL(A)}\right)$-throughput-competitive.

For the algorithm of [3], the proof gives a competitive ratio of $O(n^{3/4} \log n)$. This is the best algorithm currently known for doing collects in a model in which the adversary has complete knowledge of the current state of the system. It is likely that better algorithms are possible, even in this strong model, although the analysis of cooperative collect algorithms can be very difficult.

Other authors have devised faster algorithms for weaker models; see Section 7.3.

5.3 Proving throughput-competitiveness

We measure time by the total number of steps taken by all processes. Consider some execution, and let $C(t)$ be the set of collect operations in progress at any time t. Each collect operation consists of a sequence of atomic read and write operations; if, for some algorithm A, there is a bound $\mathrm{CL}(A)$ on the total number of read and write operations performed by collects in $C(t)$ at time t or later, this bound is called the *collective latency* of A [3].

Define the *private latency* $\mathrm{PL}(A)$ of A as the maximum number of read and write operations carried out by a single process during any one of its own collect operations.

Our progress measure keeps track of how much of the collective latency and private latency is used up by each read or write operation. It is composed of two parts for each process p: the first part, M_p, tracks steps by other processes that contribute to the collective latency of a set of collects that includes p's current collect. The second part, N_p, simply counts the number of steps done by p.

A step π at time t by a process q is *useful* for p if π is part of a collect that started before p's current collect. In a sense, π is useful if it contributes to the total work done by all collects in progress when p's current collect started. A step π is *extraneous* for p if π occurs during an interval where p has finished one collect operation but has not yet taken any steps as part of a new collect operation, and π is either the first or last operation of q in this interval. Extraneous steps do not help p in any way, but we must count them anyway for technical reasons that will become apparent in the proof

of Lemma 6.

Let $M_p(t)$ be the total number of useful and extraneous steps for p in the first t steps of the execution. Let $N_p(t)$ be the number of steps carried out by p in the first t steps of the execution.

Lemma 5 *Let A be any cooperative collect algorithm for which* $\mathrm{CL}(A)$ *and* $\mathrm{PL}(A)$ *are bounded. Then, in any execution of A in which process p completes a collect at time t, the total number of collects completed by p by time t is at least $F_p(t) - 1$, where*

$$F_p(t) = \frac{1}{2}\left(\frac{M_p(t)}{\mathrm{CL}(A) + 2(n-1)} + \frac{N_p(t)}{\mathrm{PL}(A)}\right) \tag{7}$$

Proof: Observe that F_p does not decrease over time. We will show that $F_p(t)$ rises by at most 1 during any single collect operation of p, from which the stated bound will follow.

Define $t_0 = 0$, so that $F_p(t_0) = M_p(t_0) = N_p(t_0) = 0$, and for each $i > 0$ let t_i be the time of the last step of p's i-th collect. We will bound the increase in N_p and M_p between t_i and t_{i+1} separately.

Since p performs at most $\mathrm{PL}(A)$ steps during each collect, we have $N_p(t_{i+1}) - N_p(t_i) \leq \mathrm{PL}(A)$.

Recall that M_p counts both *useful* steps for p (those operations of other processes q that occur during a collect of p and are part of a collect of q that started before the collect of p) and *extraneous* steps for p (the first and last steps of any other process q between two successive collects of p). The interval $(t_i, t_{i+1}]$ includes both an initial prefix before t starts its $(i+1)$-th collect and a suffix during which it carries out its $(i+1)$-th collect. Let s_{i+1} be the time of the first step of p's $(i+1)$-th collect. Then during the initial prefix (t_i, s_{i+1}) each other process q may carry out up to two extraneous steps, for a total of at most $2(n-1)$ extraneous steps. Useful steps occur only during the suffix $[s_{i+1}, t_{i+1}]$, and any useful step done by some process $q \neq p$, by definition, is part of a collect that has already started at time s_{i+1}. Since collects in progress at time s_{i+1} perform a total of at most $\mathrm{CL}(A)$ steps after time s_{i+1}, the total number of useful steps in the interval $(t_i, t_{i+1}]$ is at most $\mathrm{CL}(A)$. Adding together the useful steps and the extraneous steps gives $M_p(t_{i+1} - M_p(t_i) \leq \mathrm{CL}(A) + 2(n-1)$.

Thus

$$\begin{aligned} F_p(t_{i+1}) - F_p(t_i) &\leq \frac{1}{2}\left(\frac{M_p(t_{i+1}) - M_p(t_i)}{\mathrm{CL}(A) + 2(n-1)} + \frac{N_p(t_{i+1}) - N_p(t_i)}{\mathrm{PL}(A)}\right) \\ &\leq \frac{1}{2}(1+1) = 1. \end{aligned}$$

Starting with $F_p(t_0) = 0$, a simple induction argument then shows that $F_p(t_i) \leq i$ for all i.

We now exploit the fact that F_p is nondecreasing over time (which is immediate from the definitions of M_p and N_p and the fact that F_p is an increasing function of M_p and N_p). Fix some time t. Let i be the largest integer for which $F_p(t) > i$, so that we have $F_p(t) \leq i + 1$ or $i \geq F_p(t) - 1$. If $t < t_i$, then $F_p(t) \leq F_p(t_i) \leq i$. Taking the contrapositive, if $F_p(t) > i$, then $t \geq t_i$. Since t_i is defined as the completion time of p's i-th collect, p completes at least $i \geq F_p(t) - 1$ collects by time t. ■

Now we must show that our progress measure rises. It is easy to see that $\sum_p N_p$ rises by exactly 1 per step. To show that $\sum_p M_p$ rises, we partition the schedule into intervals and look at how many processors are active during each interval.

Lemma 6 *Fix some collect algorithm A. Let $(t_1, t_2]$ be any time interval, and suppose that there are exactly m processes that carry out at least one step during $(t_1, t_2]$. Then*

$$\sum_p M_p(t_2) - \sum_p M_p(t_1) \geq \binom{m}{2}. \tag{8}$$

Proof: Recall that M_p counts the total number of steps that are useful for p or extraneous for p. It is easy to see that, for any p, $M_p(t_2) - M_p(t_1) \geq 0$; this will allow us to ignore processes that do no take steps during the interval. For every *pair* of processes that both take steps during the interval, we will show that at least one step of one of the processes is either useful or extraneous for the other, and thus raises M_p by 1 for some p. This will give the desired bound by counting the number of such pairs.

Let S be the set of processes that carry out at least one step in $(t_1, t_2]$. Given distinct processes $p_1, p_2 \in S$, Define the indicator variable $m_{p_1 p_2}$ to equal 1 if p_2 takes at least one step during $(t_1, t_2]$ that is either useful or extraneous for p_1. Observe that for any $p_1 \in S$,

$$M_{p_1}(t_2) - M_{p_1}(t_1) \geq \sum_{p_2 \in S, p_2 \neq p_1} m_{p_1 p_2},$$

from which it follows that

$$\sum_p M_p(t_2) - \sum_p M_p(t_1) \quad \geq \quad \sum_{p_1 \in S} M_{p_1}(t_2) - \sum_{p_1 \in S} M_{p_1}(t_1)$$

$$\geq \sum_{p_1 \in S} \sum_{p_2 \in S, p_2 \neq p_1} m_{p_1 p_2}$$
$$= \sum_{p_1, p_2 \in S, p_1 \neq p_2} (m_{p_1 p_2} + m_{p_2 p_1}). \qquad (9)$$

We will now show that, for each distinct pair of processes p_1, p_2 in S, $m_{p_1 p_2} + m_{p_2 p_1} \geq 1$.

Let p_1 and p_2 be processes that take steps in $(t_1, t_2]$, and let C_1 and C_2 be the earliest collects of p_1 and p_2 that overlap $(t_1, t_2]$. Assume that C_1 starts before C_2, and consider the first step π of C_1 in $(t_1, t_2]$. There are three cases, depending on when π occurs relative to C_2:

1. If π occurs before C_2, then either it is the last step of C_1 that occurs before C_2, or there is some later step π' that is the last step that occurs before C_2. In either case, C_1 takes a step that is extraneous for p_2, and we have $m_{p_2 p_1} \geq 1$.

2. If π occurs during C_2, it is useful for p_2, and we again have $m_{p_2 p_1} \geq 1$.

3. If π occurs after the end of C_2, it either occurs outside of any collect, in which case it is the first operation after the end of some collect by p_2 and is extraneous for p_2, or it occurs during some later collect C_2'. In the latter case, π is useful for p_2, since C_2' starts after C_2, and thus after C_1. Whether π is extraneous for p_2 or useful for p_2, we still have $m_{p_2 p_1} \geq 1$.

We have just shown that $m_{p_2 p_1} \geq 1$ when C_1 starts before C_2. In the symmetric case where C_2 starts before C_1, a symmetric argument shows that $m_{p_1 p_2} \geq 1$. Since one of these two cases holds, we get $m_{p_1 p_2} + m_{p_2 p_1} \geq 1$ as claimed.

Since m processes carry out at least one step in $(t_1, t_2]$, there are $\binom{m}{2}$ distinct pairs of processes p_1, p_2 that each carry out at least one step in $(t_1, t_2]$. We have just shown that $m_{p_1 p_2} + m_{p_2 p_1} \geq 1$ for each such pair, and so, continuing from (9),

$$\begin{aligned} \sum_p M_p(t_2) - \sum_p M_p(t_1) &\geq \sum_{p_1, p_2 \in S, p_1 \neq p_2} (m_{p_1 p_2} + m_{p_2 p_1}) \\ &\geq \sum_{p_1, p_2 \in S, p_1 \neq p_2} 1 \\ &= \binom{m}{2}. \end{aligned}$$

■

Turning to the champion, we can trivially bound the number of collects completed during an interval of length $n-1$ by the number of active processes:

Lemma 7 *Fix some correct collect algorithm A^*. Let $t_2 = t_1 + n - 1$ and suppose that there are exactly m processes that carry out at least one step in $(t_1, t_2]$. Then A^* completes at most m collects during $(t_1, t_2]$.*

Proof: Since a process must carry out at least one step to complete a collect, the only way A^* can complete more than m collects during the interval is if some process p completes more than one collect. We will show that if this happens, there is an execution that demonstrates that A^* is *not* correct. It follows that if A^* is correct, then A^* completes at most m collects during the interval.

Suppose that there is such a process p, and let $t_p > t_1$ be the time at which p first completes a collect during $(t_1, t_2]$. Then $(t_p, t_2]$ consists of at most $n-2$ steps, and since at most one register can be read during any one step, by the Pigeonhole Principle there are exist (at least) two processes q_1, q_2 with the property that no process reads a register owned by q_1 or q_2 process during $(t_p, t_2]$. At least one of these two processes is not p; call this process q.

Let v be the value that p returns for q's register from its second collect during the interval. We will now construct a modified execution in which v is replaced by a different value v' before this collect starts. Let ξ be the execution of A^* through time t_2, and split ξ as $\xi = \alpha\beta$ where α is the prefix whose last step occurs at time t_p. Because no process reads any register owned by q in β, we can remove all steps of q in β without affecting the execution of the other processes; let β' be the result of this removal. Now construct an execution fragment γ, extending α, in which q runs in isolation until it completes its current collect, and then writes a new value $v' \neq v$ to its register. Because no process reads any register owned by q in β', the new execution $\xi' = \alpha\gamma\beta'$ is indistinguishable from ξ by any process other than q; in particular, p still returns v for q in its second collect, which starts after q writes v' in γ. Thus there is an execution in which A^* returns an incorrect value, and A^* is not a correct collect algorithm. ■

Combining Lemmas 6 and 7 gives:

Lemma 8 *Let A be a collect algorithm for which* $\mathrm{CL}(A)$ *and* $\mathrm{PL}(A)$ *are bounded, and let A^* be any collect algorithm. Let $t_2 = t_1 + n - 1$ and*

suppose m processes are active in $(t_1, t_2]$. Let F_p be the progress measure for A as defined in (7) in Lemma 5. Let C be the number of collects completed by A^* during $(t_1, t_2]$. Then

$$\begin{aligned}\frac{\sum_p (F_p(t_2) - F_p(t_1)}{C} &\geq \sqrt{\frac{n-1}{2\,\mathrm{PL}(A)\cdot(\mathrm{CL}(A)+2(n-1))}} \\ &\quad - \frac{1}{4\,(\mathrm{CL}(A)+2(n-1))}. \end{aligned} \tag{10}$$

Proof:

$$\begin{aligned}\frac{\sum_p (F_p(t_2) - F_p(t_1)}{C} &\geq \frac{\frac{1}{2}\left(\frac{\sum_p (M_p(t_2)-M_p(t_1))}{\mathrm{CL}(A)+2(n-1)} + \frac{\sum_p (N_p(t_2)-N_p(t_1))}{\mathrm{PL}(A)}\right)}{m} \\ &\geq \frac{1}{2m}\left(\frac{\binom{m}{2}}{\mathrm{CL}(A)+2(n-1)} + \frac{n-1}{\mathrm{PL}(A)}\right) \\ &= \frac{m-1}{4\,(\mathrm{CL}(A)+2(n-1))} + \frac{1}{m}\cdot\frac{n-1}{2\,\mathrm{PL}(A)}. \end{aligned} \tag{11}$$

This last quantity (11), treated as a function of m, is of the form $\frac{m-1}{a}+\frac{b}{m}$, where a and b are positive constants. Thus its second derivative is $\frac{2b}{m^3}$, which is positive for positive m. It follows that (11) is strictly convex when m is greater than 0, and thus that it has a unique local minimum (and no local maxima) in the range $m \geq 0$. This local minimum is not at $m = 0$, as the second term diverges. So it must be at some $m > 0$ at which the first derivative vanishes.

Taking the first derivative with respect to m and setting the result to 0 shows that the unique point at which the first derivative vanishes is when

$$\frac{1}{4\,(\mathrm{CL}(A)+2(n-1))} = \frac{1}{m^2}\cdot\frac{n-1}{2\,\mathrm{PL}(A)}.$$

or

$$m = \sqrt{\frac{2(n-1)\cdot(\mathrm{CL}(A)+2(n-1))}{\mathrm{PL}(A)}}. \tag{12}$$

Plugging (12) into (11) and simplifying gives the right-hand side of (10), which, as the minimum value of (11) for all m, is a lower bound on the left-hand side of (10). ■

Equation (10) effectively gives us the inverse of the competitive throughput of A, as we can sum over all intervals in the schedule and use Lemma 5

to translate the lower bound on $\sum_p F_p$ to a bound on the number of collects completed by A. Asymptotically, we can simplify (10) further by noting that $\mathrm{CL}(A)$ is always $\Omega(n)$, and that $\mathrm{PL}(A)$ is likely to be $O(n)$ for any reasonable collect algorithm A. We then get:

Theorem 9 *Let A be a collect algorithm for which $\mathrm{PL}(A) = O(n)$ and $\mathrm{CL}(A)$ is bounded. Then A is throughput-competitive with ratio $O\left(\sqrt{\mathrm{CL}(A)}\right)$.*

Proof: Fix a schedule σ of length t and a request sequence R. From Lemmas 5 and 8, we have

$$\begin{aligned}
\mathrm{done}(A,\sigma,R) &\geq \sum_p F_p(t) - n \\
&\geq \mathrm{opt}(\sigma) \cdot \sqrt{\frac{n-1}{2\,\mathrm{PL}(A) \cdot (\mathrm{CL}(A) + 2(n-1))}} \\
&\quad - \mathrm{opt}(\sigma) \cdot \frac{1}{4\,(\mathrm{CL}(A) + 2(n-1))} - n \\
&= \mathrm{opt}(\sigma) \cdot \left(\frac{1}{O(\sqrt{\mathrm{CL}(A)})} - \frac{1}{\Omega(\mathrm{CL}(A))} \right) - n \\
&= \mathrm{opt}(\sigma) \cdot \frac{1}{O(\sqrt{\mathrm{CL}(A)})} - n.
\end{aligned}$$

The last term is subsumed by the additive constant, and we are left with just the ratio $k = O\left(\sqrt{\mathrm{CL}(A)}\right)$. ■

For example, applying Theorem 9 to the collect algorithm of Ajtai *et al.* [3] gives a competitive throughput of $O(n^{3/4} \log n)$. Similarly, Aspnes and Hurwood [10] give a randomized algorithm whose collective latency is $O(n \log^3 n)$, and use an extended version of Theorem 9 to show that it has competitive throughput $O(n^{1/2} \log^{3/2} n)$.

5.4 Lower bound

It is a trivial observation that any cooperative collect algorithm has a collective latency of at least $\Omega(n)$, for the simple reason that completing even a single collect operation requires reading all n registers. It follows that Theorem 9 cannot give an upper bound on competitive throughput better than $O(\sqrt{n})$. This turns out to be an absolute lower bound on the competitive throughput of any deterministic collect algorithm, as shown in Theorem 10, below.

Theorem 10 *No deterministic algorithm for collect or write-collect has a throughput competitiveness less than* $\Omega(\sqrt{n})$.

Proof: Fix some deterministic algorithm A. We will construct a schedule in which A completes $O(\sqrt{n})$ collects, while an optimal algorithm completes $\Omega(n)$. By iterating this construction, we get an arbitrarily long schedule in which the ratio of collects completed by A to those completed by an optimal algorithm is $1/\Omega(\sqrt{n})$. Since an arbitrarily long schedule eventually overshadows any additive constant, it follows that the competitive throughput of A is at least $\Omega(\sqrt{n})$.

Choose a set $S = \{p_1, p_2, \ldots, p_m\}$ of $m = o(n)$ processes and construct a schedule σ in which these processes (and no others) take steps in round-robin order. During the first $n - m - 1$ steps of this schedule, at most $n - m - 1$ registers are read, so in particular there is some process $p \notin S$ such that no register belonging to p is read in the first $n - m - 1$ steps of the execution of A.

Extend σ to a new schedule σ' by splitting σ into segments where each processes takes two steps, and inserting $m + n + 1$ steps by p in between the first and second round of steps in each segment. The result looks like this:

$$\underbrace{p_1 p_2 \ldots p_m \overbrace{pp\ldots p}^{\times m+n+1} p_1 p_2 \ldots p_m}_{\times \left\lfloor \frac{n-m-1}{2m} \right\rfloor}.$$

This new schedule σ' is indistinguishable from σ to processes in S. So in an execution of A under σ', no process in S reads any register owned by p, and so no process in S completes a collect. Turning to p, since p can complete at most one collect for each $n-1$ steps (the minimum time to read fresh values), the number of collects completed by p during σ' is at most

$$\left\lfloor \frac{(3m + n + 1)\left\lfloor \frac{n-m-1}{2m} \right\rfloor}{n-1} \right\rfloor = O(n/m).$$

In contrast, a better A^* might proceed as follows: during each of the $\left\lfloor \frac{n-m-1}{2m} \right\rfloor$ segments of σ', first p_1 through p_m write out timestamps (and, in the case of write-collect, their inputs). Process p then gathers these timestamps in m steps (so that in can prove that the values it reads later are fresh). Process p uses n more steps to read the n registers, and writes the values of these registers, marked with the timestamps, in its last step. During the last m steps of the segment, p_1 through p_m read p's registers to

finish their collects. Thus an optimal A^* finishes at least $m+1$ collects per segment, for a total of at least $(m+1)\left\lfloor \frac{n-m-1}{2m} \right\rfloor = \Omega(n)$ collects during σ'.

So far we have mostly demonstrated the "granularity problem" that justifies the additive constant in Definition 1. To overcome this constant, we need to iterate the construction of σ', after extending it further to get A back to a state in which every process is about to start a new collect.

Observe that if a process has not yet completed a collect, it cannot do so without executing at least one operation. Let ρ_0 be the shortest schedule of the form $\sigma' pp \dots p$ such that in Algorithm A, process p has finished a collect without starting a new collect at the end of ρ_0, where p is as in the definition of σ'. Note that if p has completed all of its collects in σ', ρ_0 will be equal to σ', but in general ρ_0 will add as many as $O(n)$ additional steps by p. Note further that extending σ' to ρ_0 adds at most one additional completed collect for A.

Similarly define, for each i in the range 1 to m, ρ_i as the shortest schedule of the form $\rho_{i-1} p_i p_i \dots p_i$ such that p_i has finished a collect without starting a new collect at the end of ρ_i. As before, each such extension adds at most one additional completed collect for A, so that the total number of collects completed by A in ρ_m is at most $1+m$ more than the number completed in σ', for a total of $O\left(m + \frac{n}{m}\right)$.

This quantity is minimized when $m = \Theta(\sqrt{n})$, in which case A completes $O(\sqrt{n})$ collects during ρ_m. Because ρ_m extends σ', the number of collects completed by A^* can only increase, so A^* still completes $\Omega(n)$ collects during ρ_m.

Since at the end of ρ_m we are in a state where every process is about to start a collect, we may repeat the construction to get a sequence of phases, in each of which A completes $O(\sqrt{n})$ collects vs. $\Omega(n)$ for A^*. Call the schedule consisting of s such phases ρ^s. Then when n is sufficiently large, $\text{done}(A, \rho^s, R) \le sc\sqrt{n}$ for some constant c, while $\text{done}(A^*, \rho^s, R) \ge sc^* n$ for some constant c^*, where R is a set of request sequences consisting only of collect operations.

From Definition 1, A is k-throughput-competitive only if there exists a constant c' such that for all ρ^s,

$$\text{done}(A, \rho^s, R) + c' \ge \frac{1}{k} \text{opt}_T(\rho^s) \ge \frac{1}{k} \text{done}(A^*, \rho^s, R).$$

Applying our previous bounds on $\text{done}(A, \rho^s, R)$ and $\text{done}(A^*, \rho^s, R)$, we get

$$sc\sqrt{n} + c' \ge \frac{1}{k} sc^* n,$$

and thus

$$k \geq \frac{c^* sn}{cs\sqrt{n} + c'}.$$

Since this last inequality holds for all s, taking the limit as s goes to infinity gives

$$k \geq \frac{c^*}{c}\sqrt{n} = \Omega(\sqrt{n}).$$

■

Though we concentrate on deterministic algorithms in this paper, it is worth noting that a similar construction gives the same lower bound for randomized algorithms with an adaptive adversary. The main difference is that instead of choosing p to be the last process whose register is read, we choose p to have the highest expected time at which its register is first read, and cut off a segment when p's register is in fact read.

6 Applications

Armed with a throughput-competitive write-collect algorithm and Theorem 4, it is not hard to obtain throughput-competitive versions of many well-known shared-memory algorithms. Examples include snapshot algorithms [2, 5, 9, 12, 14],the bounded round numbers abstraction [29], concurrent timestamping systems [27, 28, 31, 33, 34, 39], and time-lapse snapshot [28]. Here we elaborate on some simple examples.

6.1 Atomic snapshots

For our purposes, 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 of Section 5.1 by 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. As with write-collect, we are combining what in some implementations may be a separate cheap operation (the update) with an expensive operation (the scan).[3]

[3] A similar combined operation appears, with its name further abbreviated to *scate*, in [14].

Snapshot objects are very useful tools for constructing more complicated shared-memory algorithms, and they have been extensively studied [2, 5, 9, 12]culminating in the protocol of Attiya and Rachman [14] which uses only $O(\log n)$ alternating writes and collects to complete a scan-update operation, giving $O(n \log n)$ total work.

We will apply Theorem 4 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 $\text{opt}_T(\sigma) \leq \text{opt}_U(\sigma)$ for any schedule σ, and so scan-update is feasible relative to write-collect.

Now let A be the Attiya-Rachman snapshot algorithm, and let B be a throughput-competitive implementation of write-collect. Let R be a set of request sequences consisting of scan-update operations. Since each process in the Attiya-Rachman snapshot algorithm completes one scan-update for every $O(\log n)$ write-collects, we have $\text{done}(B, \sigma, R_A) \leq O(\log n) \cdot \text{done}(A \circ B, \sigma, R) + O(n \log n)$, where the additive term accounts for write-collect operations that are part of scan-updates that have not yet finished at the end of σ. So we have:

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

since $frac\text{opt}_T(\sigma)\text{opt}_U(\sigma) \leq 1$. Applying Definition 2, the Attiya-Rachman snapshot is $O(\log n)$-throughput-competitive relative to write-collect. By Theorem 4, plugging in any k-throughput-competitive implementation of write-collect gives an $O(k \log n)$-throughput-competitive snapshot protocol. For example, if we use the $O(n^{3/4} \log n)$-competitive protocol of Section 5.2, we get an $O(n^{3/4} \log^2 n)$-competitive snapshot.

6.2 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 4. Let T be a an object providing the advance-collect operation, and let U be a write-collect object. Because an advance-collect must gather information from every process in the system, it implicitly contains a collect, and $\text{opt}_T(\sigma) \leq \text{opt}_U(\sigma)$ 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. Using the write-collect algorithm of Section 5.2 thus gives an $O(n^{3/4} \log n)$-competitive algorithm.

7 Conclusions

We have given a new measure for the competitive performance of distributed algorithms, which improves on the competitive latency measure of Ajtai *et al.* [3] by allowing such algorithms to be constructed compositionally. We have shown that the cooperative collect algorithm of [3] is $O(n^{3/4} \log^{3/2} n)$-competitive by this measure, from which we get an $O(n^{3/4} \log^{5/2} n)$-competitive atomic snapshot by modifying the protocol of [14], and an $O(n^{3/4} \log^{3/2} n)$-competitive bounded round numbers protocol by modifying the protocol of [29]. These modifications require only replacing the collect subroutine used in these protocols with a cooperative collect subroutine, and the proof of competitiveness does not require examining the actual working of the modified protocols in detail. We believe that a similar straightforward substitution could give competitive versions of many other distributed protocols.

We discuss some related approaches to analyzing the competitive ratio of distributed algorithms in Section 7.1. Some possible extensions of the present work are mentioned in Section 7.2.

Finally, we note that competitive ratios of $\tilde{O}(n^{3/4})$ are not very good, but they are not too much worse than Theorem 10's lower bound of $\Omega(n^{1/2})$. We describe some related work that gets closer to this bound (and, for a modified version of the problem, breaks it) in Section 7.3.

7.1 Related work

A notion related to allowing only other 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 [45] 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, 30, 35, 42–44], or a particular pattern of message delivery [13, 32, 46]. In a sense, work on optimality envisions a fundamentally different role for the adversary in which it is trying to produce bad performance for both the candidate and champion algorithms; 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 4) 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 *et al.* [21] and the distributed paging algorithm of Awerbuch *et al.* [18]. 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 compositional competitive distributed algorithms *and* does so in the natural distributed setting that permits concurrency.

7.2 Variations on competitiveness

Our work defines compositional 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 compositional 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. This leads to a general notion of *semicompetitive analysis*, which has been described in a survey paper by

the first author [7], based in part on the present work.

7.3 Improved collect algorithms

Since the appearance of the conference version of this paper, Aspnes and Hurwood [10] and Aumann [15] have shown that weakening some of the requirements of the model used here can greatly improve performance.

In particular, Aspnes and Hurwood [10] have shown that with an adversary whose knowledge of the system state is limited, collects can be performed with a near-optimal $O(n^{1/2}\log^{3/2} n)$ competitive ratio in the throughput-competitiveness model. Aumann [15] has shown that, for some applications, the freshness requirement can be weakened to allow a process to obtain a value that is out-of-date for its own collect, but that was current at the start of some other process's collect. He shows that with this weakened requirement an algorithm based on the Aspnes-Hurwood algorithm can perform collects with a competitive ratio of only $O(\log^3 n)$.

8 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, and the anonymous referees for very detailed and helpful comments on an earlier draft of this work.

References

[1] K. Abrahamson. On achieving consensus using a shared memory. In *Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, August 1988.

[2] Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *Journal of the ACM*, 40(4):873–890, September 1993.

[3] Miklos Ajtai, James Aspnes, Cynthia Dwork, and Orli Waarts. A theory of competitive analysis for distributed algorithms. In *35th Annual Symposium on Foundations of Computer Science*, pages 401–411, Santa Fe, New Mexico, 20–22 November 1994. IEEE.

[4] Noga Alon, Gil Kalai, Moty Ricklin, and Larry Stockmeyer. Lower bounds on the competitive ratio for mobile user tracking and distributed job scheduling. *Theoretical Computer Science*, 130(1):175–201, 1 August 1994.

[5] James H. Anderson. Composite registers. *Distributed Computing*, 6(3):141–154, 1993.

[6] James Aspnes. Time- and space-efficient randomized consensus. *Journal of Algorithms*, 14(3):414–431, May 1993.

[7] James Aspnes. Competitive analysis of distributed algorithms. In Amos Fiat and Gerhard Woeginger, editors, *Lecture Notes in Computer Science 1442: Proceedings of the Dagstuhl Workshop on On-Line Algorithms*, pages 118–146. Springer-Verlag, New York, NY, 1998.

[8] James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, September 1990.

[9] James Aspnes and Maurice P. Herlihy. Wait-free data structures in the asynchronous PRAM model. In *Proceedings of the 2nd Annual Symposium on Parallel Algorithms and Architectures*, pages 340–349, July 1990.

[10] James Aspnes and William Hurwood. Spreading rumors rapidly despite an adversary. *Journal of Algorithms*, 26(2):386–411, February 1998.

[11] James Aspnes and Orli Waarts. Randomized consensus in expected $O(N \log^2 N)$ operations per processor. *SIAM Journal on Computing*, 25(5):1024–1044, October 1996.

[12] Hagit Attiya, Maurice Herlihy, and Ophir Rachman. Atomic snapshots using lattice agreement. *Distributed Computing*, 8(3):121–132, 1995.

[13] Hagit Attiya, Amir Herzberg, and Sergio Rajsbaum. Optimal clock synchronization under different delay assumptions. *SIAM Journal on Computing*, 25(2):369–389, April 1996.

[14] Hagit Attiya and Ophir Rachman. Atomic snapshots in $O(n \log n)$ operations. *SIAM Journal on Computing*, 27(2):319–340, March 1998.

[15] Yonatan Aumann. Efficient asynchronous consensus with the weak adversary scheduler. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 209–218, Santa Barbara, California, 21–24 August 1997.

[16] Baruch Awerbuch and Yossi Azar. Local optimization of global objectives: Competitive distributed deadlock resolution and resource allocation. In *35th Annual Symposium on Foundations of Computer Science*, pages 240–249, Santa Fe, New Mexico, 20–22 November 1994. IEEE.

[17] Baruch Awerbuch, Yair Bartal, and Amos Fiat. Competitive distributed file allocation. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing*, pages 164–173, San Diego, California, 16–18 May 1993.

[18] Baruch Awerbuch, Yair Bartal, and Amos Fiat. Distributed paging for general networks. *Journal of Algorithms*, 28(1):67–104, July 1998.

[19] Baruch Awerbuch, Shay Kutten, and David Peleg. Competitive distributed job scheduling (extended abstract). In *Proceedings of the Twenty-Fourth Annual ACM Symposium on the Theory of Computing*, pages 571–580, Victoria, British Columbia, Canada, 4–6 May 1992.

[20] Baruch Awerbuch and David Peleg. Sparse partitions (extended abstract). In *31st Annual Symposium on Foundations of Computer Science*, volume II, pages 503–513, St. Louis, Missouri, 22–24 October 1990. IEEE.

[21] Yair Bartal, Amos Fiat, and Yuval Rabani. Competitive algorithms for distributed data management. *Journal of Computer and System Sciences*, 51(3):341–358, December 1995.

[22] Yair Bartal and Adi Rosén. The distributed k-server problem—a competitive distributed translator for k-server algorithms. *Journal of Algorithms*, 23(2):241–264, May 1997.

[23] Elizabeth Borowsky and Eli Gafni. Immediate atomic snapshots and fast renaming (extended abstract). In *Proceedings of the Twelth Annual ACM Symposium on Principles of Distributed Computing*, pages 41–51, Ithaca, New York, USA, 15–18 August 1993.

[24] Gabriel Bracha and Ophir Rachman. Randomized consensus in expected $O(n^2 \log n)$ operations. In Sam Toueg, Paul G. Spirakis, and Lefteris M. Kirousis, editors, *Distributed Algorithms, 5th International Workshop*, volume 579 of *Lecture Notes in Computer Science*, pages 143–150, Delphi, Greece, 7–9 October 1991. Springer, 1992.

[25] Benny Chor, Amos Israeli, and Ming Li. Wait-free consensus using asynchronous hardware. *SIAM Journal on Computing*, 23(4):701–712, August 1994.

[26] Danny Dolev, Ruediger Reischuk, and H. Raymond Strong. Early stopping in Byzantine agreement. *Journal of the ACM*, 37(4):720–741, October 1990.

[27] Danny Dolev and Nir Shavit. Bounded concurrent time-stamping. *SIAM Journal on Computing*, 26(2):418–455, April 1997.

[28] Cynthia Dwork, Maurice Herlihy, Serge Plotkin, and Orli Waarts. Time-lapse snapshots. *SIAM Journal on Computing*, 28(5):1848–1874, October 1999.

[29] Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Bounded round numbers. In *Proceedings of the Twelth Annual ACM Symposium on Principles of Distributed Computing*, pages 53–64, Ithaca, New York, USA, 15–18 August 1993.

[30] Cynthia Dwork and Yoram Moses. Knowledge and common knowledge in a Byzantine environment: Crash failures. *Information and Computation*, 88(2):156–186, 1990.

[31] Cynthia Dwork and Orli Waarts. Simple and efficient bounded concurrent timestamping and the traceable use abstraction. *Journal of the ACM*, 46(5):633–666, September 1999.

[32] M. J. Fischer and A. Michael. Sacrificing serializability to attain high availability of data in an unreliable network. In *Procedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 70–75, March 1982.

[33] Rainer Gawlick, Nancy Lynch, and Nir Shavit. Concurrent timestamping made simple. In Danny Dolev, Zvi Galil, and Michael Rodeh, editors, *Proceedings of the Israel Symposium on Theory of Computing and Systems (ISTCS '92)*, volume 601 of *LNCS*, pages 171–183, Berlin, Germany, May 1992. Springer.

[34] 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.

[35] Joseph Y. Halpern, Yoram Moses, and Orli Waarts. A characterization of eventual Byzantine agreement. *SIAM Journal on Computing*, 31(3):838–865, 2001.

[36] Maurice Herlihy. Randomized wait-free concurrent objects (extended abstract). In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, pages 11–21, Montreal, Quebec, Canada, 19–21 August 1991.

[37] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[38] Amos Israeli and Ming Li. Bounded time-stamps. *Distributed Computing*, 6(4):205–209, 1993.

[39] Amos Israeli and Meir Pinhasov. A concurrent time-stamp scheme which is linear in time and space. In Adrian Segall and Shmuel Zaks, editors, *Distributed Algorithms, 6th International Workshop, WDAG '92*, volume 647 of *Lecture Notes in Computer Science*, pages 95–109, Haifa, Israel, 2–4 November 1992. Springer.

[40] Lefteris M. Kirousis, Paul Spirakis, and Philippas Tsigas. Simple atomic snapshots: A linear complexity solution with unbounded time-stamps. *Information Processing Letters*, 58(1):47–53, 8 April 1996.

[41] Elias Koutsoupias and Christos H. Papadimitriou. Beyond competitive analysis. *SIAM Journal on Computing*, 30(1):300–317.

[42] Yoram Moses and Mark R. Tuttle. Programming simultaneous actions using common knowledge. *Algorithmica*, 3:121–169, 1988.

[43] G. Neiger and R. Bazzi. Using knowledge to optimally achieve coordination in distributed systems. *Theoretical Computer Science*, 220(1):31–65, 1999.

[44] Gil Neiger and Mark R. Tuttle. Common knowledge and consistent simultaneous coordination. *Distributed Computing*, 6(3):181–192, 1993.

[45] Christos H. Papadimitriou and Mihalis Yannakakis. Linear programming without the matrix (extended abstract). In *Proceedings of the Twenty-Fifth Annual ACM Symposium on the Theory of Computing*, pages 121–129, San Diego, California, 16–18 May 1993.

[46] Boaz Patt-Shamir and Sergio Rajsbaum. A theory of clock synchronization. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on the Theory of Computing*, pages 810–819, Montréal, Québec, Canada, 23–25 May 1994.

[47] Yaron Riany, Nir Shavit, and Dan Touitou. Towards a practical snapshot algorithm. *Theoretical Computer Science*, (269):163–201, 2001.

[48] Michael Saks, Nir Shavit, and Heather Woll. Optimal time randomized consensus—making resilient algorithms fast in practice. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 351–362, San Francisco, California, 28–30 January 1991.

[49] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, February 1985.

[50] Paul M. B. Vitányi and Baruch Awerbuch. Atomic shared register access by asynchronous hardware (detailed abstract). In *27th Annual Symposium on Foundations of Computer Science*, pages 233–243, Toronto, Ontario, Canada, 27–29 October 1986. IEEE. See also errata appearing in 28th FOCS.