Competitive Analysis of Distributed Algorithms

James Aspnes*

Yale University, Department of Computer Science

Abstract. Most applications of competitive analysis have involved on-line problems where a candidate *on-line* algorithm must compete on some input sequence against an optimal *off-line* algorithm that can in effect predict future inputs. Efforts to apply competitive analysis to fault-tolerant distributed algorithms require accounting for not only this *input nondeterminism* but also *system nondeterminism* that arises in distributed systems prone to asynchrony and failures. This paper surveys recent efforts to adapt competitive analysis to distributed systems, and suggests how these adaptations might in turn be useful in analyzing a wider variety of systems. These include tools for building competitive algorithms by composition, and for obtaining more meaningful competitive ratios by limiting the knowledge of the off-line algorithm.

1 Introduction.

Like on-line algorithms, distributed algorithms must deal with limited information and unpredictable user and system behavior. Unlike on-line algorithms, in many distributed algorithms the primary source of difficulty is the possibility that components of the underlying system may fail or behave badly. In a distributed system, processes may crash, run at wildly varying speeds, or execute erroneous code; messages may be lost, garbled, or badly delayed. As a consequence, the worst-case performance of many algorithms can be very bad, and may have little correspondence to performance in more typical cases.

It is not surprising that the technique of **competitive analysis** [ST85] should be useful for taming the excesses of worst-case analysis of distributed algorithms. Of course, new opportunities and complications arise in trying to apply competitive analysis directly to fault-tolerant distributed systems. Distributed systems have a natural split between the inputs coming from the users above and the environment provided by the system below— by exploiting this split it is possible, among other things, to build competitive algorithms by composition. On the other hand, because a distributed system consists of many individual components with limited information, one must be careful in how one defines the powers of the "off-line" algorithm so that the boundaries between these separate components do not become blurred. Work in this area has

* 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

yielded several useful techniques for carefully controlling how much information the off-line algorithm is allowed to use.

Section 2 discusses distributed systems in general. Section 3 describes competitive analysis in its traditional form. Some variants on traditional competitive measures that are useful in models in which the nondeterminism naturally splits into two categories are described in Section 4. For these *semicompetitive* performance measures, composition of algorithms is possible, as described in Section 5. Examples of applications of these techniques to distributed problems are given in Sections 6 and 7. Finally, Section 8 discusses directions in which this area might profitably be extended.

A caveat: this survey is concerned primarily with examples in the distributed algorithms literature of applying competitive analysis directly to nondeterminism in the underlying system. No attempt is made to cover the vast body of excellent work on on-line problems, such as distributed paging, load-balancing, routing, mobile user tracking, etc., that arise in networks and other distributed systems.

2 Distributed Systems.

There are many forms of distributed systems, and a wide variety of theoretical models used to study them. However, there are two properties that show up in most distributed system models, which distinguish them from uniprocessor or parallel systems. The first and most important property is that a distributed system consists of more than one process, which may represent a real physical CPU, or might correspond to an abstract entity like a process or thread in a timesharing system. The second property is that the multiple processes of a distributed system are poorly coordinated— each runs its own program; the communication channels between them may be slow, expensive, and unreliable; and individual processes may crash, become faulty, or run at varying speeds. It is this poor coordination that distinguishes distributed systems from parallel systems, in which processes typically execute the same program in very close synchrony with each other using a powerful and reliable communication system. The line between parallel and distributed systems is not a sharp one, but a reasonable rule of thumb is that parallel systems are predictable; the same program with the same input running on the same parallel machine should give essentially the same results every time it is run. In contrast, distributed systems are riddled with nondeterminism— distributed algorithms are always at the mercy of an unreliable and sometimes hostile infrastructure, and must strive for robustness and consistency despite it.

Because of this inherent nondeterminism and the difficulty of communication, an individual process in a distributed system is likely to have only an incomplete and often out-of-date picture of the state of the system as a whole. Like an on-line algorithm, it must still make decisions, which may turn out to be suboptimal later, based on this limited information. This property of distributed algorithms gives an on-line flavor to most problems that arise in distributed computing, and

suggests that techniques such as competitive analysis that have been useful for dealing with the unpredictable request sequences of on-line problems should be useful for dealing with the unpredictable system behavior of distributed problems. However, it is not immediately obvious how on-line techniques can best be adapted to distributed systems.

The complication is that for many distributed algorithms we can distinguish between two different sources of nondeterminism. The first source is the user-supplied **input** to the algorithm: those distributed algorithms that carry out inherently on-line tasks such as scheduling or load balancing must suffer the whims of unpredictable users just as much as any on-line algorithm running on a single processor. However, these distributed algorithms must also contend with system nondeterminism, often summarized as a **schedule** that specifies when processes can take steps, what order messages are received in, and so forth. This schedule is just as unpredictable (and is often assumed to be just as adversarial) as the input.

3 Competitive Analysis.

Competitive analysis [ST85] compares the performance of a general-purpose algorithm given particular inputs against specialized algorithms optimized for each of those inputs. From an abstract perspective, one considers a class of algorithms $\mathcal{A}$, a class of inputs $\mathcal{I}$, and a **cost function** $\mathrm{cost}(A, I)$ that assigns a non-negative cost to each algorithm A and input I. A particular algorithm A is **k-competitive** if there exists a constant c such that for all inputs I in $\mathcal{I}$ and algorithms A^* in A,

$$\mathrm{cost}(A, I) \leq k \cdot \mathrm{cost}(A^*, I) + c.$$

If $c = 0$, the algorithm is said to be **strictly k-competitive**. The minimum value of k for which an algorithm is k-competitive is the algorithm's **competitive ratio**. If $c = 0$, this quantity is just the maximum over all I of $\mathrm{cost}(A, I)/\min_{A^* \in \mathcal{A}} \mathrm{cost}(A^*, I)$.

The competitive ratio of an algorithm measures its performance on a particular input relative to all other algorithms. Often it is convenient to speak of the optimal algorithm for a particular input. When analyzing on-line algorithms, there is no harm in assuming that the best algorithm is an **off-line algorithm**— one that can predict the input— as the ability to choose an optimal algorithm based on the input is equivalent to using single algorithm with such predictive powers. In contrast, the algorithm being evaluated can be described as the **on-line algorithm** because it does not have such powers.

For distributed algorithms, the assumption that an algorithm can predict the input can have surprising consequences, and it is necessary in some models to limit the class of algorithms $\mathcal{A}$ to exclude full clairvoyance. With such a limitation, it is less misleading to use the neutral term **champion** to refer to the optimal algorithm for a particular input, since this term does not suggest any

special knowledge that the champion might have. Similarly, the term **candidate** provides a neutral way to refer to the algorithm that is being evaluated.[2]

Payoffs Instead of Costs. One can also define the competitiveness of an algorithm whose performance is measured in terms of a non-negative payoff function payoff(A, I). The idea here is that for some problems it is more natural to think in terms of how much can be accomplished given a fixed set of resources than how many resources are needed to accomplish a fixed set of tasks. In terms of payoffs, an algorithm is k-competitive if there exists a constant c such that for all I in $\mathcal{I}$ and A^* in A,

$$\text{payoff}(A, I) + c \geq \frac{1}{k} \text{payoff}(A^*, I).$$

Note that the competitive ratio is inverted for consistency with the cost version; smaller competitive ratios imply better performance with either measure.

4 Semicompetitive Analysis

For distributed algorithms, it often makes sense to assume that the cost function depends on three parameters: the algorithm chosen, an input (which may or may not be common to both the candidate and champion algorithm) and an **environment** (which is always assumed to be common to both the candidate and champion algorithm). A problem is then defined in terms of a set of algorithms $\mathcal{A}$, a set of inputs $\mathcal{I}$, and a set of environments $\mathcal{E}$, together with a cost function cost(A, I, E) that assigns a non-negative cost to each algorithm A, input I, and environment E. From an abstract perspective it is not necessary to worry about how the nondeterminism in a system is split into the input and the environment, though in practice it may strongly affect how a problem looks.

One effect of such split nondeterminism is that it allows what we will call **semicompetitive analysis**, in which a candidate algorithm is measured against an optimal champion running with the same environment but not necessarily the same input. The goal is still to be able to measure the performance of a particular general-purpose candidate algorithm relative to the best specialized algorithms in some class. However, there are now several possible measures depending on how the inputs are chosen for the candidate and champion algorithms.

Common Inputs. If one assumes that the input and environment are the same for both candidate and champion, one has the traditional definition of k-competitiveness. An algorithm A is **traditionally k-competitive** if there exists a constant c such that for all A^* in $\mathcal{A}$, I in $\mathcal{I}$, and E in $\mathcal{E}$,

$$\text{cost}(A, I, E) \leq k \cdot \text{cost}(A^*, I, E) + c.$$

[2] These terms were suggested by [AADW94].

Worst-Case vs. Best-Case Inputs. A more useful measure arises if one assumes that the inputs are not shared between the candidate and the champion. Let us define an algorithm A in a split-nondeterminism framework as **k-competitive** if there exists a constant c such that for all A^* in $\mathcal{A}$, I and I^* in $\mathcal{I}$, and E in $\mathcal{E}$,

$$\text{cost}(A, I, E) \leq k \cdot \text{cost}(A^*, I^*, E) + c.$$

In effect, this definition assumes that the candidate and champion share the same environment, but that the candidate faces a worst-case input and the champion a best-case input. The assumption that the candidate and champion do not have the same inputs— the only difference between this definition and traditional k-competitiveness— has far-reaching consequences. In particular it allows the modular construction of competitive algorithms, as described below in Section 5.

The definition of k-competitiveness given above is based on the "throughput model" of [AW96]. The use of the term "k-competitiveness" for this measure is justified by several practical considerations. First, the semicompetitive definition is stronger than the traditional definition. Not only is a k-competitive algorithm a traditionally k-competitive algorithm (immediate from inspection of the quantifiers), but if one also assumes that the set of inputs consists of a single point or that the input has no effect on the cost of an algorithm, the new definition reduces to the traditional definition. Second, in many respects the new measure captures the intuitive notion of competitiveness at least as well as the traditional measure in those situations where one can reasonably assume that only part of the nondeterminism affecting an algorithm is likely to be shared between it and an optimal algorithm (for example, when considering a subroutine whose input, in the form of procedure calls, is supplied by a higher-level candidate algorithm). Finally, there is little chance of confusion between the new and old definitions, as the new definition applies only in the case of split nondeterminism, and the presence of such a split will usually be obvious from context.

Worst-Case vs. Worst-Case Inputs. Yet another useful measure is obtained if it is assumed that both the candidate and champion algorithms face worst-case inputs. Define an algorithm A to be **k-optimal** if there exists a constant c such that for all A^* in $\mathcal{A}$, I in $\mathcal{I}$, and E in $\mathcal{E}$, there exists an input I^* in $\mathcal{I}$ such that

$$\text{cost}(A, I, E) \leq k \cdot \text{cost}(A^*, I^*, E) + c.$$

The difference between this definition and the previous one lies only in the choice of quantifiers. In effect, the input I^* given to the champion algorithm A^* is a worst-case input, chosen to maximize the champion's cost. Consequently, k-optimality is a weaker notion than k-competitiveness; however, it may still be a useful measure in contexts where k-competitiveness tells us little about the actual performance of an algorithm. An example is given in Section 7.

The definition of k-optimality given above was suggested by [AW96] by analogy to their semicompetitive definition of k-competitiveness. It also corresponds very closely to an earlier measure used by Patt-Shamir and Rajsbaum in their work on clock synchronization [PSR94].

4.1 Relations Between the Measures

There is a nice relationship between semicompetitive analysis and traditional competitive analysis:

Theorem 1. *If an algorithm is k-competitive, then it is traditionally k-competitive. If an algorithm is traditionally k-competitive, it is k-optimal.*

Proof. Immediate from the definitions. □

In particular, if one has an upper bound on the competitive ratio of an algorithm (in the split-nondeterminism sense), one immediately gets an upper bound on the traditional competitive ratio of an algorithm. Similarly, a lower bound on optimality implies a lower bound on competitiveness. Thus even if one is not interested in the semicompetitive measures directly, they may still provide a useful tool for bounding the traditional competitive ratio of an algorithm.

4.2 Payoffs Instead of Costs

Just as one can define the competitiveness of an algorithm in the traditional sense in terms of payoffs instead of costs, one can do the same for the semicompetitive measures. Let us assume as above that we have a class of algorithms $\mathcal{A}$ against which the candidate algorithm will compete, a set of inputs $\mathcal{I}$ and a set of environments $\mathcal{E}$. Suppose further that we have a payoff function assigning a non-negative value $\text{payoff}(A, I, E)$ to each A in $\mathcal{A}$, I in $\mathcal{I}$, and E in $\mathcal{E}$.

An algorithm is traditionally k-competitive if there exists a constant c such that for all A^* in $\mathcal{A}$, I in $\mathcal{I}$, and E in $\mathcal{E}$,

$$\text{payoff}(A, I, E) + c \geq \frac{1}{k} \cdot \text{payoff}(A^*, I, E).$$

An algorithm is k-competitive if there exists a constant c such that for all A^* in $\mathcal{A}$, I and I^* in $\mathcal{I}$, and E in $\mathcal{E}$,

$$\text{payoff}(A, I, E) + c \geq \frac{1}{k} \cdot \text{payoff}(A^*, I^*, E). \tag{1}$$

Finally, an algorithm is k-optimal if there exists a constant c such that for all A^* in $\mathcal{A}$, I in $\mathcal{I}$, and E in $\mathcal{E}$, there exists an input I^* in $\mathcal{I}$ such that

$$\text{payoff}(A, I, E) + c \geq \frac{1}{k} \cdot \text{payoff}(A^*, I^*, E).$$

It is not difficult to see that k-competitiveness implies traditional k-competitiveness which in turn implies k-optimality in the payoff model just as in the cost model.

5 Modularity

One of the fundamental tools in traditional algorithm design is the ability to construct algorithms by composition. However, competitive analysis in general appears to forbid such modular constructions 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. This lack of modularity impedes the development of practical competitive algorithms.

Fortunately, by treating the input to a subroutine separately from its environment, it is possible to recover the ability to compose algorithm while retaining the advantages of competitive analysis. The key is the notion of **relative competitiveness** defined by [AW96]. Relative competitiveness is a measure of how well an algorithm A uses a competitive subroutine B, which takes into account not only how large A's cost is relative to B but also whether or not the decision to use B was a good idea in the first place. To do so, it considers the relative performance of three distinct executions: an execution of the combined algorithm $A \circ B$; an execution of an optimal A^* (which may or may not use a subroutine corresponding to B); and an execution of an optimal B^*. This approach requires operating within a semicompetitive framework, as described in Section 4, in which the input and environment of an algorithm are treated separately. The details of the definition of relative competitiveness are given below in Section 5.1.

The remarkable property of relative competitiveness is that it acts like a traditional worst-case performance measure when composing algorithms together. Glossing over some small technical details, if an algorithm A is l-competitive relative to a subroutine B, and B is itself k-competitive, then the combined algorithm $A \circ B$ is kl-competitive, even when compared against algorithms that do not use B or anything like B. This is exactly the same multiplicative effect that one gets with traditional worst-case analysis, where a parent algorithm that calls a subroutine l times at a cost of k units per call pays kl total cost. Except that here we are looking at competitive ratios.

The details of how one can compose competitive algorithms in this fashion are given below in Section 5.2.

5.1 Relative Competitiveness

Formally, it is assumed that we have two sets of inputs and environments. Algorithm A takes an input from $\mathcal{I}_A$ and an environment from $\mathcal{E}_A$; similarly, B takes an input from $\mathcal{I}_B$ and an environment from $\mathcal{E}_B$. In the composite algorithm $A \circ B$ the input to B is provided by its parent routine A— one can think of this input as the procedure calls given to B. Similarly, the environment of A is provided by its subroutine B— one can think of this environment as the return values from these procedure calls. The input to A and the environment of B are provided by the adversary.

In the executions of A^* and B^* the situation is not quite symmetric. The difference arises from the fact that A^* and B^* are completely independent algorithms; and while B^* is in effect an optimal version of B, A^* is not an optimal version of A but instead is an optimal version of $A \circ B$. The effect of this difference is that while B^* takes an input from $\mathcal{I}_B$ and an environment from $\mathcal{E}_B$, just as B does, A^* takes an input from $\mathcal{I}_A$ but its environment comes from $\mathcal{E}_B$.

To complete the picture, it is necessary to have two cost measures. The first, cost_{AB}, measures the cost of algorithms that see inputs from $\mathcal{I}_A$ and environments from $\mathcal{E}_B$; it will be applied to both the composite algorithm $A \circ B$ and the champion A^*. The second, cost_B, measures the cost of algorithms that see inputs from $\mathcal{I}_B$ and environments from $\mathcal{E}_B$; it will be applied to B (as a component of $A \circ B$) and B^*. To avoid difficulties with division by zero, it is convenient to require that cost_B always be positive.

The definition is as follows. Given $\mathcal{A}_A$, $\mathcal{A}_B$, $\mathcal{I}_A$, $\mathcal{I}_B$, $\mathcal{E}_A$, $\mathcal{E}_B$, cost_{AB}, and cost_B, an algorithm A is l-competitive relative to an algorithm B if there exists a constant c such that for all E in $\mathcal{E}_B$,

$$\max_{I_A \in \mathcal{I}_A} \frac{\text{cost}_{AB}(A \circ B, I_A, E) - c}{\text{cost}_B(B, I_B, E)} \leq l \cdot \frac{\min_{A^* \in \mathcal{A}_A, I_A^* \in \mathcal{I}_A} \text{cost}_{AB}(A^*, I_A^*, E)}{\min_{B^* \in \mathcal{A}_B, I_B^* \in \mathcal{I}_B} \text{cost}_B(B^*, I_B^*, E)}. \quad (2)$$

Note that the input I_B to B is the input generated by A when run in interaction with B. If the constant c is zero, then A is **strictly** l-competitive relative to B.

It is sometimes possible to show that an algorithm A is l-competitive relative to *any* algorithm in a class of algorithms $\mathcal{A}_B$. In this case we we write that A is l-competitive relative to $\mathcal{A}_B$. The competitiveness of an algorithm relative to the class of all correct subroutines for solving a particular problem is often a more useful measure than its competitiveness relative to a particular subroutine— it implies that one can substitute a more efficient subroutine for a less efficient one and get an improvement in performance.

The payoff version of relative competitiveness is defined analogously as follows. Given $\mathcal{A}_A$, $\mathcal{A}_B$, $\mathcal{I}_A$, $\mathcal{I}_B$, $\mathcal{E}_A$, $\mathcal{E}_B$, payoff_{AB}, and payoff_B, an algorithm A is l-competitive relative to an algorithm B if there exists a constant c such that for all E in $\mathcal{E}_B$,

$$\max_{I_A \in \mathcal{I}_A} \frac{\text{payoff}_{AB}(A \circ B, I_A, E) + c}{\text{payoff}_B(B, I_B, E)} \geq \frac{1}{l} \cdot \frac{\max_{A^* \in \mathcal{A}_A, I_A^* \in \mathcal{I}_A} \text{payoff}_{AB}(A^*, I_A^*, E)}{\max_{B^* \in \mathcal{A}_B, I_B^* \in \mathcal{I}_B} \text{payoff}_B(B^*, I_B^*, E)}.$$

Note that the sign of the additive constant c has changed, in order to be consistent with the definition of k-competitiveness. Again, it is required that payoff_B is never zero; however, in the payoff model it is in fact possible to drop this requirement without causing too many difficulties (for details see [AW96]).

It is also possible to define relative optimality for both the cost and payoff models. For relative optimality, the inputs I_A^* and I_B^* are chosen to maximize the costs (minimize the payoffs) of A^* and B^*; there are no other differences.

5.2 The Composition Theorem

The usefulness of relative competitiveness is captured in Composition Theorem of [AW96]. The theorem as given here is taken more-or-less directly from [AW96], where it was first stated in a slightly less general form. The version given here has been adapted slightly to fit into the more general framework used in this survey. To simplify the statement of the theorem, let us assume throughout this section that $\mathcal{A}_A$, $\mathcal{A}_B$, $\mathcal{I}_A$, $\mathcal{I}_B$, $\mathcal{E}_A$, $\mathcal{E}_B$ and either cost_{AB} and cost_B or payoff_{AB} and payoff_B are fixed.

Theorem 2 (Composition Theorem). *Let A be l-competitive relative to B, B k-competitive, and suppose that there exists a non-negative constant c such that for all E in $\mathcal{E}_E$, either*

$$\min_{I_A^* \in \mathcal{I}_A, A^* \in \mathcal{A}_A} \text{cost}_{AB}(A^*, I_A^*, E) \geq c \min_{I_B^* \in \mathcal{I}_B, B^* \in \mathcal{A}_B} \text{cost}_B(B^*, I_B^*, E) \tag{3}$$

or

$$\max_{I_A^* \in \mathcal{I}_A, A^* \in \mathcal{A}_A} \text{payoff}_{AB}(A^*, I_A^*, E) \leq c \max_{I_B^* \in \mathcal{I}_B, B^* \in \mathcal{A}_B} \text{payoff}_B(B^*, I_B^*, E) \tag{4}$$

(as appropriate). Then $A \circ B$ is kl-competitive.

Proof. The proof given here is adapted from [AW96].

Let us consider only the case of costs; the case of payoffs is nearly identical and in any case has been covered elsewhere [AW96]. The proof involves only simple algebraic manipulation, but it is instructive to see where the technical condition (3) is needed.

To avoid entanglement in a thicket of superfluous parameters let us abbreviate the cost of each algorithm X with the appropriate input and environment as $C(X)$, so that $\text{cost}_{AB}(A \circ B, I_A, E)$ becomes $C(A \circ B)$, $\text{cost}_B(B, I_B, E)$ becomes $C(B)$, $\text{cost}_{AB}(A^*, I_A^*, E)$ becomes $C(A^*)$, and $\text{cost}_B(B^*, I_B^*, E)$ becomes $C(B^*)$.

Fix E. The goal is to show that $C(A \circ B) \leq kl \cdot C(A^*) + c_{AB}$, where C_{AB} does not depend on E. We may assume without loss of generality that A^* and its input I_A^* are chosen to minimize $C(A^*)$, and that the input I_A is chosen to maximize $C(A)$. So in particular $C(A^*$ is equal to the left-hand side of inequality (3) and the numerator of the right-hand side of inequality (2). Similarly, choose B^* and I_B^* to minimize $C(B^*)$.

Thus (2) can be rewritten more compactly as

$$\frac{C(A \circ B) - c_{AB}}{C(B)} \leq l \cdot \frac{C(A^*)}{C(B^*)},$$

where c_{AB} is the constant from (2). Multiplying out the denominators gives

$$(C(A \circ B) - c_{AB})\, C(B^*) \leq l \cdot C(A^*)C(B). \tag{5}$$

Similarly, the k-competitiveness of B means that

$$C(B) \leq k \cdot C(B^*) + c_B, \tag{6}$$

where c_B is an appropriate constant.

Plugging (6) into the right-hand side of (5) gives

$$\begin{aligned}(C(A \circ B) - c_{AB})\, C(B^*) &\leq l \cdot C(A^* \cdot (k \cdot C(B^*) + c_B) \\ &= kl \cdot C(A^*)C(B^*) + l \cdot C(A^*)c_B.\end{aligned}$$

Dividing both sides by $C(B^*)$ and moving c_{AB} gives

$$C(A \circ B) \leq kl \cdot C(A^*) + c_{AB} + lc_B \frac{C(A^*)}{C(B^*)}. \tag{7}$$

The last term is bounded by a constant if (3) holds; thus $A \circ B$ is kl-competitive. □

Much of the complexity of the theorem, including the technical conditions (3) and (4), is solely a result of having to deal with the additive constant in the k-competitiveness of B. If one examines the last steps of the proof, it is evident that the situation is much simpler if B is strictly competitive.

Corollary 3. *Let A be l-competitive relative to B and let B be strictly k-competitive. Then $A \circ B$ is kl-competitive. If in addition A is strictly competitive relative to B, $A \circ B$ is strictly kl-competitive.*

Proof. Consider the proof of Theorem 2. Since B is strictly competitive, c_B in (7) is zero, and the technical condition (3) is not needed. If A is strictly competitive, then c_{AB} is zero as well, implying $A \circ B$ is strictly competitive. A similar argument shows that the corollary also holds in the payoff model. □

Note that the Composition Theorem and its corollary can be applied transitively: if A is competitive relative to $B \circ C$, B is competitive relative to C, and C is competitive, then A is competitive (assuming the appropriate technical conditions hold).

6 Example: The Wait-Free Shared Memory Model

In this section we describe some recent approaches to applying competitive analysis to problems in the **wait-free shared memory model** [Her91]. In this model, a collection of n processes communicate only indirectly through a set of single-writer atomic registers. A protocol for carrying out some task or sequence of tasks is **wait-free** if each process can finish its current task regardless of the relative speeds of the other processes. Timing is under the control of an adversary scheduler that is usually modeled as a function that chooses, based on the current state of the system, which process will execute the next operation. This adversary is under no restrictions to be fair; it can, for example, simulate up to $n-1$ process crashes simply by choosing never to schedule those processes again.

The wait-free shared memory model is a natural target for competitive analysis. The first reason is that many of the algorithms that are known for this

model pay for their high resilience (tolerating up to $n-1$ crashes and arbitrary asynchrony) with high worst-case costs. Thus there is some hope that competitive analysis might be useful for showing that in less sever cases these algorithms perform better. The second reason is that the absence of restrictions on the adversary scheduler makes the mathematical structure of the model very clean. Thus it is a good jumping off point for a more general study of the applicability of competitive analysis to the fault-tolerant aspects of distributed algorithms. Finally, a third reason is that under the assumption of single-writer registers, there is a natural problem (the *collect problem*) that appears implicitly or explicitly in most wait-free shared-memory algorithms. By studying the competitive properties of this problem, we can learn about the competitive properties of a wide variety of algorithms.

Sections 6.1 and 6.2 describe the collect problem and solution to it without regard to issues of competitiveness. Section 6.3 discusses why competitiveness is a useful tool for analyzing the performance of collect algorithms. The following sections describe how it has been applied to such algorithms and the consequences of doing so.

Some of the material in theses sections is adapted from [AADW94], [AW96], and [AH96].

6.1 Collects: A Fundamental Problem

When a process starts a task in the wait-free model, it has no means of knowing what has happened in the system since it last woke up. Thus to solve almost all non-trivial problems a process must be able to carry out a **collect**, an operation in which it learns some piece of information from each of the other processes.

The collect problem was first abstracted by Saks, Shavit, and Woll [SSW91]. The essential idea is that each process owns a single-writer multi-reader atomic register, and would like to be able carry out write operations on its own register and **collect operations** in which it learns the values in all the registers. The naive method for performing a collect is simply to read all of the registers directly; however, this requires at least $n-1$ read operations (n if we assume that a process does not remember the value in its own register). By cooperating with other processes, it is sometimes possible to reduce this cost by sharing the work of reading all the registers.

There are several versions of the collect problem. The simplest is the "one-shot" collect, in which the contents of the registers are fixed and each process performs only one collect. The one-shot version of the problem will not be discussed much in this paper, but it is worth noting some of the connections between one-shot collects and other problems in distributed computing. Without the single-writer restriction, the one-shot collect would be isomorphic to the problem of Certified Write-All, in which n processes have to write to n locations while individually being able to certify that all writes have occurred (indeed, the collect algorithm of Ajtai et al. [AADW94] is largely based on a Certified Write-All algorithm of Anderson and Woll [AW91]). One can also think of the collect problem as an asynchronous version of the well-known **gossip problem** [EM89],

in which n persons wish to distribute n rumors among themselves with a minimum number of telephone calls; however, in the gossip problem, which persons communicate at each time is fixed in advance by the designer of the algorithm; with an adversary controlling timing this ceases to be possible.

The more useful version of the problem, and the one that we shall consider here, is the **repeated collect** problem. This corresponds exactly to simulating the naive algorithm in which a process reads all n registers to perform a collect. An algorithm for repeated collects must provide for each process a *write* procedure that updates its register and a *collect procedure* that returns a vector of values for all of the registers. Each of these procedures may be called repeatedly. The values returned by the collect procedure must satisfy a safety property called **regularity** or **freshness**: any value that I see as part of the result of a collect must be **fresh** in the sense that it was either present in the appropriate register when my collect started or it was written to that register while my collect was in progress.

The repeated collect problem appears at the heart of a wide variety of shared-memory distributed algorithms (an extensive list is given in [AADW94]). What makes it appealing as a target for competitive analysis is that the worst-case performance of any repeated collect algorithm is never better than that of the naive algorithm. The reason for this is the strong assumptions about scheduling. In the wait-free shared-memory model, a process cannot tell how long it may have been asleep between two operations. Thus when a process starts a collect, it has no way of knowing whether the register values have changed since its previous collect. In addition, the adversary can arrange that each process does its collects alone by running only one process at a time. If the adversary makes these choices, any algorithm must read all of the $n-1$ registers owned by the other processes to satisfy the safety property.

But there is hope: this lower bound applies to any algorithm that satisfies the safety property— including algorithms that are chosen to be optimal for the timing pattern of a particular execution. Thus it is plausible that a competitive approach would be useful.

6.2 A Randomized Algorithm for Repeated Collects

Before jumping into the question of how one would apply competitive analysis to collects, let us illustrate some of the issues by considering a particular collect algorithm, the "Follow the Bodies" algorithm of [AH96].

As is often the case in distributed computing, one must be very precise about what assumptions one makes about the powers of the adversary. The Follow-the-Bodies algorithm assumes that a process can generate a random value and write it out as a single atomic operation. This assumption appears frequently in early work on consensus; it is the "weak model" of Abrahamson [Abr88] and was used in the consensus paper of Chor, Israeli, and Li [CIL94]. In general, the weak model in its various incarnations permits much better algorithms (e.g., [AB96, Cha96]) for such problems as consensus than the best known algorithms in the more fashionable "strong model" (in which the adversary can stop a process in

between generating a random value and writing it out). The assumption that the adversary cannot see coin-flips before they are written is justified by an assumption that in a real system failures, page faults, and similar disastrous forms of asynchrony are likely to be affected by *where* each process is reading and writing values but not by *what* values are being read or written. For this reason the adversary in the weak model is sometimes called *content-oblivious*.

Even with a content-oblivious adversary the collect problem is still difficult. Nothing prevents the adversary from stopping a process between reading new information from another process's register and writing that information to its own register. Similarly the adversary can stop a process between making a random choice of which register to *read* and the actual read operation. (This rule corresponds to an assumption that not all reads are equal; some might involve cache misses, network delays, and so forth.)

This power of the adversary turns out to be quite important. Aspnes and Hurwood show that an extremely simple algorithm works in the restricted case where the adversary cannot stop a read selectively depending on its target. Each process reads registers randomly until it has all the information it needs. However, the adversary that *can* stop selected reads can defeat this simple algorithm simple algorithm by choosing one of the registers to be a "poison pill": any process that attempts to read this register will be halted. Since on average only one out of every n reads would attempt to read the poisonous register, close to n^2 reads would be made before the adversary would be forced to let some process actually swallow the poison pill.

In order to avoid this problem, in the Follow-the-Bodies algorithm a process leaves a note saying where it is going before attempting to read a register.[3] Poison pills can thus be detected easily by the trail of corpses leading to them. The distance that a process will pursue this trail will be $\lambda \ln n$, where λ is constant chosen to guarantee that the process reaches its target with high probability.

Figure 1 depicts one pass through the loop of the resulting algorithm. The description assumes that each process stores in its output register both the set of values S it has collected so far and its *successor*, the process it selected to read from most recently.

- Set p to be a random process, and write out p as our successor.
- Repeat $\lambda \ln n$ times:
  - Read (S', p') from the register of p.
  - Set S to be the union of S and S'.
  - Set p to p'.
  - Write out the new S and p.

Fig. 1. The Follow-the-Bodies Algorithm (One Pass Only)

[3] It is here that the assumption that one can flip a coin and write the outcome atomically is used.

For the moment we have left out the termination conditions for the loop, as they may depend on whether one is trying to build a one-shot collect or a repeated collect (which requires some additional machinery to detect fresh register values).

The Follow-the-Bodies algorithm has the property that it spreads information through the processes' registers rapidly regardless of the behavior of the adversary. In [AH96] it is shown that:

Theorem 4. *Suppose that in some starting configuration the register of each process p contains the information K_p and that the successor fields are set arbitrarily. Fix $\lambda \geq 9$, and count the number of operations carried out by each process until its register contains the union over all p of K_p, and write W for the sum of these counts.*

$$\Pr W \geq 37\lambda^2 n \ln^3 n \leq \frac{2}{n^{\lambda-5}}.$$

Here the rapidity of the spread of information is measured in terms of total work. Naturally, the adversary can delay any updates to a particular process's register for an arbitrarily long time by putting that process to sleep; what the theorem guarantees is that (a) when the process wakes up, it will get the information it needs soon (it is likely that by then the first process it looks at will have it); and (b) the torpor of any individual process or group of processes cannot increase the total work of spreading information among their speedier comrades.

The proof of the theorem is a bit involved, and the interested reader should consult [AH96] for details. In the hope of making the workings of the algorithm more clear we mention only that the essence of the proof is to show that after each phase consisting of $O(n \ln n)$ passes through the loop, the size of the smallest set of processes that collectively know all of the information is cut in half. After $O(\ln n)$ of these phases some process knows everything, and it is not long before the other processes read its register and learn everything as well. (The extra log factor comes from the need to read $\lambda \ln n$ registers during each pass through the loop).

In [AH96] it is shown how to convert this rumor-spreading algorithm to a repeated-collect algorithm by adding a simple timestamp scheme. Upon starting a collect a process writes out a new timestamp. Timestamps spread through the process's registers in parallel with register values. When a process reads a value *directly* from its original register, it tags that value by the most recent timestamp it has from each of the other processes. Thus if a process sees a value tagged with its own most recent timestamp, it can be sure that that value was present in the registers after the process started its most recent collect, i.e. that the value is fresh.

The resulting algorithm is depicted in Figure 2. Here, S tracks the set of values (together with their tags) known to the process. The array T lists each process's most recent timestamps. Both S, T, and the current successor are periodically written to the process's output register.

- Choose a new timestamp τ and set our entry in T to τ.
- While some values are unknown:
  - Set p to be a random process, write out p as our successor and T as our list of known timestamps.
  - Repeat $\lambda \ln n$ times:
    - Read the register of p. Set S to be the union of S and the values field. Update T to include the most recent timestamps for each process. Set p to the successor field.
    - Write out the new S and T.
- Return S.

Fig. 2. Repeated Collect Algorithm

The performance of this algorithm is characterized by its **collective latency** [AADW94], an upper bound on the total amount of work needed to complete all collects in progress at some time t:[4]

Theorem 5. *Fix a starting time t. Fix $\lambda \geq 9$. Each process carries out a certain number of steps between t and the time at which it completes the collect it was working on at time t. Let W be the sum over all processes of these numbers. Then*

$$\Pr W \geq 74\lambda^2 n \ln^3 n \leq \frac{4}{n^{\lambda-5}}.$$

Proof. Divide the steps contributing to W into two classes: (i) steps taken by processes that do not yet know timestamps corresponding to all of the collects in progress at time t; and (ii) steps taken by processes that know all n of these timestamps. To bound the number of steps in class (i), observe that the behavior of the algorithm in spreading the timestamps during these steps is equivalent to the behavior of the non-timestamped algorithm. Similarly, steps in class (ii) correspond to an execution of the non-timestamped algorithm when we consider the spread of values tagged by all n current timestamps. Thus the total time for both classes of steps is bounded by twice the bound from Theorem 4, except for a case whose probability is at most twice the probability from Theorem 4. □

6.3 Competitive Analysis and Collects

Theorem 5 shows that the Follow-the-Bodies algorithm improves in one respect on the naive collect algorithm. For the naive algorithm, the collects in progress at any given time may take n^2 operations to finish, while with high probability Follow-the-Bodies finishes them in $O(n \log^3 n)$ operations. Unfortunately, this

[4] The "collective" part of "collective latency" refers not to the fact that the algorithm is doing collect operations but instead to the fact that the latency it is measuring is a property of the group of processes as a whole.

improvement does not translate into better performance according to traditional worst-case measures.

As noted above, if a process is run in isolation it must execute at least $n-1$ operations to carry out a collect, no matter what algorithm it runs, as it can learn fresh values only by reading them directly. Under these conditions the naive algorithm is optimal. Even if one considers the amortized cost of a large number of collects carried out by different processes, the cost per collect will still be $\Omega(n)$ if no two processes are carrying out collects simultaneously. Yet in those situations where the processes have the opportunity to cooperate, Theorem 5 implies that they can do so successfully, combing their efforts so that in especially good cases (where a linear number of processes are running at once) the expected amortized cost of a single collect drops to $O(\log^3 n)$.

It is clear that algorithms like Follow-the-Bodies improve on the naive collect. But how can one quantify this improvement in a way that is meaningful outside the limited context of collect algorithms? Ideally, one would like to have a measure that recognizes that distributed algorithms may be run in contexts where many or few processes participate, where some processes are fast and some are slow, and where the behavior of processes may vary wildly from one moment to the next. Yet no simple parameter describing an execution (such as the number of active processes) can hope to encompass such detail.

Competitive analysis can. By using the performance of an optimal, specialized algorithm as a benchmark, competitive analysis provides an objective measure of the difficulty of the environment (in this case, the timing pattern of an asynchronous system) in which a general-purpose algorithm operates. To be competitive, the algorithm must not only perform well in worst-case environments; its performance must adapt to the ease or difficulty of whatever environment it faces. Fortunately, it is possible to show that many collect algorithms have this property.

Some complications arise. In order to use competitive analysis, it is necessary to specify precisely what is included in the environment— intuitively, on what playing field the candidate and champion algorithms will be compared. An additional issue arises because assuming that the champion has complete knowledge of the environment (as the hypothetical "off-line" algorithm has when computing the competitive ratio of an on-line algorithm) allows it to implicitly communicate information from one process to another at no cost that a real distributed algorithm would have to do work to convey. Neither difficulty is impossible to overcome. In the following sections we describe two measures of competitive performance for distributed algorithms that do so.

6.4 A Traditional Approach: Latency Competitiveness

The competitive latency model of Ajtai et al. [AADW94] uses competitive analysis in its traditional form, in which all nondeterminism in a system is under the control of the adversary and is shared between both the candidate and champion algorithms. In the context of the repeated collect problem, it is assumed that the adversary controls the execution of an algorithm by generating (possibly

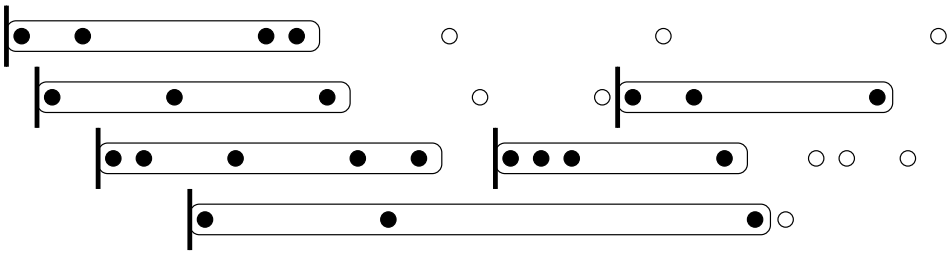

Fig. 3. Latency model. New high-level operations (ovals) start at times specified by the adversary (vertical bars). Adversary also specifies timing of low-level operations (small circles). Cost to algorithm is number of low-level operations actually performed (filled circles).

in response to the algorithm's behavior) a schedule that specifies when collects start and when each process is allowed to take a step (see Figure 3). A process halts when it finishes a collect; it is not charged for opportunities to take a step in between finishing one collect and starting another (intuitively, we imagine that it is off doing something else). The **competitive latency** of a candidate algorithm is the least constant k, if any, that guarantees that the expected total number of operations carried out by the candidate on a given schedule σ is at most k times the cost of an optimal distributed algorithm running on the same schedule (modulo an additive constant).

In terms of the general definition of competitive analysis from Section 3, $\mathcal{E}$ consists of all schedules as defined above. The set of algorithms $\mathcal{A}$ consists of all distributed algorithms that satisfy a correctness condition. The cost measure is just the number of operations required by a particular algorithm. For the collect problem, this correctness condition requires that an algorithm guarantee that the values returned are fresh regardless of the schedule. In particular, while processes in a champion algorithm are allowed to use their implicit knowledge of the schedule to optimize their choices of what registers to read and how to cooperate with one another, they cannot use this knowledge to avoid verifying the freshness of the values they return.

There is a general principle involved here that points out one of the difficulties of applying competitive analysis directly to distributed algorithms. The difficulty is that if one is not careful, one can permit the processes in the champion algorithm to communicate with each other at zero cost by virtue of their common knowledge of the schedule. For example: consider a schedule in which no process ever writes to a register. If the champion is chosen after the schedule is fixed, as occurs whenever we have an off-line adversary, then it can be assumed that every process in the champion implicitly knows the schedule. In particular, each process in the champion can observe that no processes are writing to the registers and thus that the register values never change. So when asked to perform a collect, such a process can return a vector consisting of the initial values

of the registers at zero cost. Needless to say, a situation in which the champion pays zero cost is not likely to yield a very informative competitive ratio.

What happened in this example is that if one assumes that the champion processes all have access to the schedule, each champion process can then compute the entire state of the system at any time. In effect, one assumes that the champion algorithm is a global-control algorithm rather than a distributed algorithm. It is not surprising that against a global-control algorithm it is difficult to be competitive, especially for a problem such as collect in which the main difficulty lies in transmitting information from one process to another.

On the other hand, notice that the champion algorithm in this case is not correct for all schedules; any schedule in which any register value changes at some point will cause later collects to return incorrect values. Limiting the class of champion algorithms to those whose correctness does not depend on operating with a specific schedule restores the distributed character of the champion algorithm. Yet while such a limit it prevents pathologies like zero-cost collects, it still allows the champion to be "lucky", for example by having all the processes choose to read from the one process that happens in a particular execution to have already obtained all of the register values. Thus the reduction in the champion's power is limited only to excluding miraculous behavior, but not the surprising cleverness typical of off-line algorithms.

With this condition, Ajtai et al. show that if an algorithm has a maximum collective latency of L at all times, then its competitive ratio in the latency model is at most $L/n + 1$. Though this result is stated only for deterministic algorithms, as observed in [AH96] it can be made to apply equally well to randomized algorithms given a bound L on the expected collective latency.

The essential idea of the proof in [AADW94] of the relationship between collective latency and competitive latency is to divide an execution into segments and show that for each such segment, the candidate algorithm carries out at most $L + n$ operations and the champion carries out at least n operations. The lower bound on the champion is guaranteed by choosing the boundaries between the segments so that in each segment processes whose collects start in the segment are given at least n chances to carry out an operation; either they carry out these n operations (because their collects have not finished yet) or at least n read operations must have been executed to complete these collects (because otherwise the values returned cannot be guaranteed to be fresh). For an algorithm with expected collective latency L, the cost per segment is at most $L + n$; n for the n steps during the segment, plus at most an expected L operations to complete any collects still in progress at the end of the segment. By summing over all segments this gives a ratio of $L/n + 1$.

For the Follow-the-Bodies algorithm, the expected value of L is easily seen to be $O(n \log^3 n)$ (from Theorem 5). It follows that:

Theorem 6. *The competitive latency of the Follow-the-Bodies algorithm is* $O(\log^3 n)$.

It is worth noting that this result is very strong; it holds even against an *adaptive off-line* adversary [BDBK+90], which is allowed to choose the champion

algorithm after seeing a complete execution of the candidate. In contrast, the best known lower bound is $\Omega(\log n)$ [AADW94].

It is still open whether or not an equally good deterministic algorithm exists. The best known deterministic algorithm, from [AADW94], has a competitive latency of $O(n^{1/2} \log^2 n)$.

6.5 A Semicompetitive Approach: Throughput Competitiveness

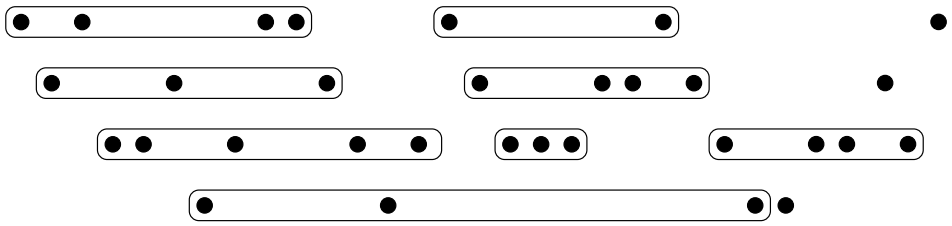

Fig. 4. Throughput model. New high-level operations (ovals) start as soon as previous operations end. Adversary controls only timing of low-level operations (filled circles). Payoff to algorithm is number of high-level operations completed.

A semicompetitive approach of the sort discussed in Section 4 is used in the **competitive throughput** model of Aspnes and Waarts [AW96]. In this model, the adversary no longer controls the starting time of collects; instead, both the candidate and the champion try to complete as many collects as possible in the time available (see Figure 4). It also distinguishes between the environment, which takes the form of a schedule that now specifies only when low-level operations such as reads and writes occur; and the input, which takes the form of a specification of what sequence of high-level operations each process is to perform but not when it must perform them. (For the collect problem the input is generally not very interesting, since the collect algorithm can only perform one kind of high-level operation.)

Formally, the model can be viewed as using the semicompetitive definition of competitiveness from Section 4. The set of algorithms $\mathcal{A}$ consists of all correct distributed algorithms; as in the preceding section, correctness must be defined independently of any single schedule to avoid pathologies. The set of environments $\mathcal{E}$ consists of all schedules defining the timing of low-level operations. The set of inputs $\mathcal{I}$ consists of all sequences of requests to perform high-level tasks. The competitive throughput model is a payoff model; competitiveness is defined in terms of a payoff function $\text{payoff}(A, I, E)$ that measures *how many* of the high-level tasks in I can be completed by A given the schedule E. The **throughput competitiveness** of an algorithm A is the least value k for which the payoff of A is always within a factor of k of the payoff of an optimal A^* (modulo an additive constant).

The motivation for adopting a semicompetitive measure like throughput competitiveness is that it allows competitive algorithms to be constructed modularly as described in Section 5. This is particularly important for collect algorithms since collect appears as a subroutine in such a large number of other algorithms. If one can show that these algorithms are throughput competitive relative to collect, then the only thing needed to make them throughput competitive in their own right is to plug in a throughput-competitive collect subroutine.

In order to do this, a small technical complication must first be dealt with. No interesting algorithm performs only collects; for information to pass from one process to another somebody must be writing to the registers. So instead of using a subroutines that performs only collects one uses a subroutine that supports two operations: the collect operation as described above, and a **write-collect**, consisting of a write operation followed by a collect. The intent is to simulate an object that provides both writes and collects. However, because throughput competitiveness assumes a worst-case choice of operations for the candidate to perform, but a best-case choice for the champion, it is important that all operations supported by the subroutine have roughly the same granularity. Otherwise it becomes possible for the candidate to be forced to carry out only expensive operations (like collects) while the champion carries out only cheap operations (like writes).[5] Fortunately, in practice in almost all algorithms that use writes and collects, each process performs at most one write in between each pair of collects, and so one can treat such algorithms as using write-collect and collect operations. Furthermore, since all known cooperative collect algorithms start each collect with a low-level write operation (in order to write out a new timestamp), it costs nothing to modify such an algorithm to perform write-collects as well as collects.

It turns out that the same bound on collective latency— the total number of operations required to complete all collects in progress at any given time— used to show the latency competitiveness of a collect algorithm can also be used to show throughput competitiveness. A second condition is also needed; there must be a bound on the **private latency**— the number of operations required to complete any *single* collect. This second condition can easily be guaranteed by dovetailing a collect algorithm that guarantees low collective latency with the naive algorithm that guarantees low private latency by having a process read all

[5] In principle, there are other ways to avoid this difficulty. For example, one could define the payoff function so that completed collect operations produced more value than completed write operations; but then it might be difficult to show that an algorithm that performed both writes and collects to achieve some higher goal is in fact competitive relative to this mixed-payoff subroutine. Alternatively, one could use the notion of k-optimality (where the champion also faces a worst-case input) instead of k-competitiveness. However, k-optimality is weaker than traditional k-competitiveness, so results about the performance of higher-level algorithms obtained using the k-optimality of the collect subroutine would not immediately imply competitiveness of the higher-level algorithms in the traditional sense. Still, there are not yet very many examples in the literature of the use of these techniques, so it remains to be seen which approach is likely to be most useful in the long run.

n registers itself.

With this modification, and the earlier modification to add write-collect operations, the Follow-the-Bodies algorithm from Section 6.2 has an collective latency of $O(n \log^3 n)$ (on average) and a private latency of at most $2n$. A general bound on throughput competitiveness based on these two parameters is given in [AW96]; plugging in the values for Follow-the-Bodies gives:

Theorem 7. *The competitive throughput of the Follow-the-Bodies algorithm is* $O(n^{1/2} \log^{3/2} n)$.

The proof of this bound is quite complex and it would be difficult even to try to summarize it here. Its overall structure is similar to the technique for bounding latency competitiveness described in Section 6.4, in that it proceeds by dividing an execution into segments and proving a lower bound on the candidate's payoff and an upper bound on the champion's payoff for each segment. The mechanism used for this division is a potential function taking into account many fine details of the execution. The interested reader can find the complete argument in the full version of [AW96].

It might be surprising that the competitive ratio for essentially the same algorithm is much larger in the throughput model than in the latency model. The reason for the difference is the increased restrictions placed on the champion by the latency model. Because the champion can only start collects when the candidate does, it is not possible for the champion to take advantage of a particularly well-structured part of the schedule to finish many collects using its superior knowledge while the candidate wanders around aimlessly. Instead, the most the champion can hope to do is finish quickly the same collects that the candidate will finish more slowly— but the candidate will not finish these collects too much more slowly if it uses an algorithm with low collective latency. By contrast, in the throughput model it is possible to construct schedules in which the champion has a huge advantage. An example (taken from [AW96]) is given in the following section.

6.6 Lower Bound on Throughput Competitiveness of Collect

To illustrate some of the difficulties that arise in the throughput model, consider the following theorem and its proof:

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

Proof. (This proof is taken directly from [AW96].)

Because of the additive term in the definition of competitive ratio, it is not enough to exhibit a single schedule on which a given collect algorithm fails. Instead, we will describe a randomized adversary strategy for producing schedules of arbitrary length on which the ratio between the number of collects performed by a given candidate algorithm and the number of collects performed by an optimal champion tends to $\Theta\sqrt{n}$.

The essential idea is that we will build up an arbitrarily-long schedule out of phases. In each phase, most of the work will be done by a randomly-chosen "patsy". Profiting from the patsy's labors, in the champion algorithm, will be $\sqrt{n}$ "active" processes. 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.

Let us be more specific. The adversary fixes the $\sqrt{n}$ active processes at the start of the schedule, and chooses a new patsy uniformly at random from the set of all non-active processes at the start of each phase. Each phase consists of one or more rounds. In each round, first each active process takes one step, then the patsy takes $n + \sqrt{n} + 1$ steps, and finally each active process takes a second step. If, during any round, some active process reads the patsy's register or learns all n register values, the adversary cleans up by running each active process and the patsy in an arbitrary order until each finishes its current collect. A new phase then starts with a new patsy, chosen at random independently of previous choices.

In the champion algorithm, each active process writes a timestamp to its register at the start of each round. The patsy then reads these timestamps ($\sqrt{n}$ steps), reads all the registers (n steps), and writes a summary of their contents, together with the timestamps, to its own output register (1 step). Finally, each active process reads the patsy's register. The result is that in each round, the champion algorithm completes $\sqrt{n} + 1$ collects. To simplify the analysis, we will assume that the champion algorithm does no work at all during the clean-up stage and the end of a phase. Under this assumption the champion completes $k(\sqrt{n} + 1)$ collects during a k-round phase.

In the candidate algorithm, each active process completes exactly one collect during each phase. Since no active process reads the patsy's register until the last round of the phase, the patsy cannot use any values obtained by the active processes prior to the last round. Thus the patsy completes at most one collect for every n operations that it executes prior to the last round, plus at most two additional collects during the last round and the clean-up stage. Since the patsy executes $n + \sqrt{n} + 1$ operations per round, the total number of collects it completes during a k-round phase is thus at most $k(1 + \frac{1}{\sqrt{n}} + \frac{1}{n}) + 2$, and the total number of collects completed by all processes during a k-round phase is at most $k(1 + \frac{1}{\sqrt{n}} + \frac{1}{n}) + 2 + \sqrt{n}$.

It remains only to determine the expected number of rounds in a phase. A phase can end in two ways: either some process finds the patsy, or some process learns all n values. Since the patsy is chosen at random, finding it requires an expected $(n - \sqrt{n} + 1)/2$ reads and at most $n - \sqrt{n}$ reads. Learning all n values requires n reads of the registers. Thus finding the patsy is a better strategy for the active processes, and since they together execute $2\sqrt{n}$ operations per round, they can find it in an expected $(\sqrt{n} - 1)/4$ rounds.

Plugging this quantity in for k gives an expected $(n-1)/4$ collects per phase for the champion algorithm and at most an expected $\frac{\sqrt{n}-1/n}{4} + 2 + \sqrt{n} \leq 2\sqrt{n}$ collects per phase for the candidate. In the limit as the number of phases goes

to infinity, the ratio between the payoff to the candidate and the payoff to the champion goes to the ratio of these expected values, giving a lower bound on the competitive ratio of $\Omega(\sqrt{n})$. □

6.7 Competitive Collect as a Subroutine

The importance of collect is that it appears as a subroutine in many wait-free distributed algorithms. The value of showing that a particular collect algorithm is competitive in the throughput model is that one can then apply the Composition Theorem (Theorem 2) to show that many of these algorithms are themselves competitive when they are modified to use a competitive collect. Several examples of this approach are given in [AW96]; one simple case is reproduced below, in order to make more concrete how one might actually show that an algorithm is competitive relative to some class of subroutines.

A **snapshot** algorithm, like a collect algorithm, simulates an array of n single-writer registers. It supports a **scan-update** operation that writes a value to one of the registers (this part is the "update") and then returns a snapshot, which is a vector of values for all of the registers (this part is the "scan"). Unlike a collect algorithm, which must satisfy only the requirement that all values returned by a collect are at least as recent as the start of that collect, the snapshot algorithm must guarantee that different snapshots appear to be instantaneous. In practice, this means that process P cannot be shown an older value than process Q in one register and a newer value in a different register.

There are many known wait-free algorithms for snapshot (a list is given in [AW96]). The best algorithm currently known in terms of the asymptotic work required for a single snapshot is the algorithm of Attiya and Rachman [AR93], in which each process uses $O(\log n)$ alternating writes and collects in order to complete a single scan-update operation. (The reason why a scan-update requires several rounds of collects is that the processes must negotiate with each other to ensure that the snapshots they compute are consistent.) If the collects are implemented using the naive algorithm that simply reads all n registers directly, this bound translates into a cost of $O(n \log n)$ primitive operations per collect in the worst case. However, by plugging in a competitive collect algorithm, it is possible to improve on this bound in many executions.

In order to do so it is first necessary to argue that the Attiya-Rachman algorithm is competitive relative to a subroutine providing write-collect and collect operations as described in Section 6.5. To remain consistent with the definition given in Section 5.1, which requires nonzero payoffs for the subroutine, we will consider only schedules in which the candidate collect algorithm used as subroutine in Attiya-Rachman completes at least one collect.[6] We would like to show that for a suitable choice of l, for any such schedule E, that there exists a

[6] As noted in Section 5.1, for payoff models like the throughput model this requirement is not strictly necessary.

constant c such that

$$\max_{I_A \in \mathcal{I}_A} \frac{\text{payoff}_{AB}(A \circ B, I_A, E) + c}{\text{payoff}_B(B, I_B, E)} \geq \frac{1}{l} \cdot \frac{\max_{A^* \in \mathcal{A}_A, I_A^* \in \mathcal{I}_A} \text{payoff}_{AB}(A^*, I_A^*, E)}{\max_{B^* \in \mathcal{A}_B, I_B^* \in \mathcal{I}_B} \text{payoff}_B(B^*, I_B^*, E)} \tag{8}$$

holds, where A is the Attiya-Rachman snapshot, B is an arbitrary collect algorithm, and $\mathcal{I}_A$, $\mathcal{I}_B$, etc., are defined appropriately.

It is easy to get an upper bound of one on the ratio on the right-hand side. Because a single scan-update operation can be used to simulate a single write-collect or collect, the payoff (i.e., number of scan-updates completed) of an optimal snapshot algorithm cannot exceed the payoff (number of collects and/or write-collects completed) of an optimal collect algorithm. For the left-hand side, the Attiya-Rachman algorithm completes one scan-update for every $O(\log n)$ write-collects done by any single process. So if c is set to n (to account for partially completed scan-updates), then

$$\frac{\text{payoff } AB(A \circ B) + n}{\text{payoff}(B)} \geq \frac{1}{O(\log n)}.$$

It follows that (8) holds for $l = O(\log n)$.

To use Theorem 2, it is also necessary to show that the technical condition (4) holds. But this just say that the maximum number of scan-updates that can be completed in a given schedule is at most a constant times the maximum number of write-collects. As observed above, this constant is one, and so Theorem 2 implies that plugging Follow-the-Bodies ($O(n^{1/2} \log^{3/2} n)$-competitive) into Attiya-Rachman ($O(\log n)$-competitive relative to collect) gives an $O(n^{1/2} \log^{5/2} n)$-competitive snapshot algorithm.

Observe that in the above argument, essentially no properties of the Attiya-Rachman snapshot algorithm were used except (a) the fact that scan-update is a "stronger" operation than either collect or write-collect, and (b) the fact that Attiya-Rachman uses $O(\log n)$ write-collects per scan-update. It is not unusual to be able to treat a parent algorithm as a black box in this way in traditional worst-case analysis— but this is one of the first examples of being able to do the same thing while doing competitive analysis. In addition, since almost any operation is stronger than write-collect in the wait-free model, a similar argument works for a wide variety of algorithms that use collects. Some additional examples are given in [AW96].

7 Example: Guessing Your Location in a Metric Space

So far we have seen examples of analyzing distributed problems using both the traditional definition of competitiveness and the semicompetitive definition of competitiveness from Section 4. For some problems neither approach is appropriate, and instead the notion of k-optimality gives more information about the actual merits of an algorithm.

One such problem can be described simply as guessing one's location in a metric space based on locally available information. Imagine that you have been placed by an adversary somewhere in a metric space (say, the surface of the Earth). You have the ability to observe the local landscape, and you would like to be able to make as close a guess as possible to your actual location based on what you see. In some locations (at the base of the Eiffel Tower; in a store that displays its wide selection of Global Positioning System boxes), the best possible guess is likely to be quite accurate. In others (somewhere in the Gobi desert; inside a sealed cardboard box), the best possible guess is likely to be wildly off. Yet it is clear that some algorithms for making such guesses are better than others.

How can one measure the performance of an algorithm for this problem? A traditional competitive approach might look something like this: the actual location x is the environment that is shared between the candidate and champion. Both candidate and champion are shown some view v, a function $f(x)$ of the location. The cost of a guess y is just the distance between y and x.

Unfortunately, in this framework, the optimal champion always guesses x correctly— at a cost of zero— while the candidate must choose among the many possible y for which $f(y) = f(x)$. Unless the candidate is lucky enough also to hit x exactly (which the adversary can prevent if it is only moderately clever), the competitive ratio of any algorithm is infinite.

No improvement comes from moving to a semicompetitive approach. Suppose that we assume that it is the observed view v that is shared between candidate and champion, but that the actual locations x and x^* may be any two points for which $f(x) = f(x^*) = v$. More formally, let us use the semicompetitive definition of k-competitiveness with the shared environment being the view and the differing inputs being the actual locations of the candidate and champion. Unfortunately, once again the champion (which is given a best-case input) guesses its location exactly, and unless the subset of the metric space that produces the appropriate view is very small indeed, the candidate once again has an infinite competitive ratio.

The solution is to use k-optimality. As above, let us assume that the the view v is shared between both candidate and champion, the actual locations x and x^* of the candidate and champion may be distinct; but in this case, suppose that *both* candidate and champion are given worst-case locations. Now the cost to the champion of choosing y^* will be the distance between y^* and the most distant x^* for which $f(x^*) = v$. If the candidate algorithm chooses any point y for which $f(y) = v$, then from the triangle inequality $d(y, x) \leq d(y, y^*) + d(y^*, x^*) \leq 2d(y^*, x^*)$, and so any candidate algorithm that is smart enough makes a guess consistent with what it sees will be 2-optimal. Particularly clever algorithms (say, those which always guess a *center* of the set of points consistent with the observed data) can be as good as 1-optimal. Though the range of optimality ratios is not large, it does give us a way to distinguish exceptionally stupid algorithms (worse than 2-optimal) from plausible algorithms (2-optimal) from good algorithms (1-optimal).

This example has been abstracted almost to the point of triviality, but it does illustrate issues that have been studied in the context of real distributed systems. The example is inspired by work on clock synchronization by Patt-Shamir and Rajsbaum [PSR94]; in this work, the goal of a clock synchronization algorithm is to make a good estimate of the offsets between different clocks in a distributed system based on data piggybacked on the messages sent by some underlying algorithm. For some communication patterns (e.g., ones in which no messages are exchanged), it is impossible to make a very accurate estimate—but the possibility of using traditional competitive analysis is foreclosed by the ability of an off-line algorithm to guess the correct offsets exactly. Their solution was to use a notion of 1-optimality, which the definition in Section 4 generalizes.

Of course, how one actually computes a good estimate of the offsets between clocks in a distributed system based on limited information is a much more complicated problem than simply picking a good point somewhere in the middle of the appropriate metric space. The interested reader will find algorithms for solving several interesting variants of the problem in [PSR94].

8 Conclusions

The preceding sections have concentrated on how the techniques of competitive analysis have been useful for analyzing fault-tolerant distributed algorithms, where the primary source of difficulty is not unpredictable inputs but unpredictable behavior in the underlying system. This approach is still quite new, and there are many questions that have yet to be considered. Not only may it be possible to apply competitive analysis to a wider variety of distributed models, but the adaptations needed to fit competitive analysis to the models described above may be useful for analyzing problems that do not necessarily involve distributed systems. There are three main areas of research suggested by this work.

Extensions to other models of distributed computation. The models described in Section 6 depend on a number of details of the wait-free shared-memory model. This model is a good testbed for new measures, since the lack of restraints on the behavior of the adversary makes the model relatively simple from a formal perspective; however, in many cases other models are more realistic. These include models in which there are much tighter controls on the timing of events (for example, those in which synchrony or fairness conditions hold or in which a small upper bound exists on the number of failures), models with different communications primitives, and models in which not all processes can be trusted to carry out the algorithm correctly. Very little has been done to study how competitive analysis can be used to analyze how algorithms respond to hostile system behavior in these models.

Non-distributed applications of relative competitiveness. Relative competitiveness and the composition theorem were defined in [AW96] to analyze a class of problems that arise in a particular model of distributed systems. However,

as the authors of [AW96] point out, when viewed abstractly there is nothing about relative competitiveness that depends on having a distributed system. The notion of relative competitiveness opens up the possibility of constructing competitive algorithms for a wide variety of problems by composition: building them up one subroutine at a time, with the competitiveness of each subroutine, proved separately, contributing to the competitiveness of the whole. With this ability, in addition to looking for good superproblems, like k-servers or Metrical Task Systems, that subsume many on-line problems, it may be fruitful to look for good subproblems whose solutions can be used as subroutines in many relative-competitive algorithms.

Limiting the clairvoyance of the off-line algorithm. Competitive analysis is not very useful for "lady or the tiger" problems (like the example in Section 7) in which an off-line algorithm can always guess correctly information that no on-line algorithm could ever hope to obtain. In many such cases it is known from real-world constraints that no algorithm of any kind could deduce the correct information. When this is true, it makes sense to limit the knowledge of the off-line algorithm to what might reasonably be available in the real-world situation being modeled. However, traditional competitive analysis provides no mechanism for doing so. [7] Here, a semicompetitive approach using k-optimality may be useful. Information that the champion algorithm is allowed to use on can be made part of the environment shared with the candidate. Information that the champion algorithm is not allowed to use can be made part of the input that is assumed to be worst-case for both champion and candidate.

9 Acknowledgments

Some of the material in this survey has been adapted from [AADW94], [AW96], and [AH96]. None of this work would have been possible without the efforts of my co-authors Miklos Ajtai, Cynthia Dwork, Will Hurwood, and Orli Waarts.

[7] The technique of **comparative analysis** [KP94] might seem to be the right answer to these problems. Comparative analysis measures the value of information by comparing the best algorithms in two "information regimes" (e.g., paging algorithms with l-lookahead versus paging algorithms with no lookahead). A complication, from the point of view of someone trying to construct algorithms, is that the definition of comparative analysis assumes that the algorithm in the weaker class is chosen only after the algorithm in the stronger class is known. This does not prevent the stronger algorithm from knowing at birth bizarrely detailed properties of the request sequence (for example, a paging algorithm might know that if page 127 is requested first, the remainder of the request sequence includes twice as many even as odd-numbered pages); it just means that some hypothetical weaker algorithm may know these properties too. Thus comparative analysis is the opposite of what we want: instead of taking away the implausible knowledge of the off-line algorithm, it merely adds to the implausible knowledge of its on-line competitor.

References

[AADW94] M. Ajtai, J. Aspnes, C. Dwork, and O. Waarts. A theory of competitive analysis for distributed algorithms. In *Proc. 35th Symp. of Foundations of Computer Science*, pages 401–411, 1994.

[AB96] Y. Aumann and M.A. Bender. Efficient asynchronous consensus with the value-oblivious adversary scheduler. In *Proc. 23rd International Colloquium on Automata, Languages, and Programming*, 1996.

[Abr88] K. Abrahamson. On achieving consensus using a shared memory. In *Proceedings of the Seventh ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 291–302, August 1988.

[AH96] J. Aspnes and W. Hurwood. Spreading rumors rapidly despite an adversary. In *Proc. 15th ACM Symposium on Principles of Distributed Computing*, pages 143–151, 1996.

[AR93] H. Attiya and O. Rachman. Atomic snapshots in $o(n \log n)$ operations. In *Proc. 12th ACM Symposium on Principles of Distributed Computing*, pages 29–40, 1993.

[AW91] R. Anderson and H. Woll. Wait-free parallel algorithms for the union-find problem. In *Proc. 23rd ACM Symposium on Theory of Computing*, pages 370–380, 1991.

[AW96] J. Aspnes and O. Waarts. Modular competitiveness for distributed algorithms. In *Proc. 28th ACM Symposium on the Theory of Computing*, pages 237–246, 1996.

[BDBK+90] S. Ben-David, A. Borodin, R. Karp, G. Tardos, and A. Widgerson. On the power of randomization in on-line algorithms. In *Proc. 22nd Symposium on Theory of Algorithms*, pages 379–386, 1990.

[Cha96] T. Chandra. Polylog randomized wait-free consensus. In *Proc. 15th ACM Symposium on Principles of Distributed Computing*, 1996.

[CIL94] B. Chor, A. Israeli, and M. Li. Wait–free consensus using asynchronous hardware. *SIAM J. Comput.*, 23(4):701–712, August 1994. Preliminary version appears in *Proceedings of the 6th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing,* pages 86-97, 1987.

[EM89] S. Even and B. Monien. On the number of rounds needed to disseminate information. In *Proc. 1st ACM Symposium on Parallel Algorithms and Architectures*, 1989.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[KP94] E. Koutsoupias and C. Papadimitriou. Beyond competitive analysis. In *Proc. 25th Symposium on Foundations of Computer Science*, pages 394–400, 1994.

[PSR94] B. Patt-Shamir and S. Rajsbaum. A theory of clock synchronization. In *Proc. 26th ACM Symposium on the Theory of Computing*, pages 810–819, 1994.

[SSW91] 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, 1991.

[ST85] D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.

This article was processed using the LaTeX macro package with LLNCS style