Algorithmica (1994) 11: 542–571

Algorithmica

Competitive Randomized Algorithms for Nonuniform Problems

A. R. Karlin,[1] M. S. Manasse,[1] L. A. McGeoch,[2] and S. Owicki[1]

Abstract. Competitive analysis is concerned with comparing the performance of on-line algorithms with that of optimal off-line algorithms. In some cases randomization can lead to algorithms with improved performance ratios on worst-case sequences. In this paper we present new randomized on-line algorithms for snoopy caching and the spin-block problem. These algorithms achieve competitive ratios approaching $e/(e-1) \approx 1.58$ against an oblivious adversary. These ratios are optimal and are a surprising improvement over the best possible ratio in the deterministic case, which is 2. We also consider the situation when the request sequences for these problems are generated according to an unknown probability distribution. In this case we show that deterministic algorithms that adapt to the observed request statistics also have competitive factors approaching $e/(e-1)$. Finally, we obtain randomized algorithms for the 2-server problem on a class of isosceles triangles. These algorithms are optimal against an oblivious adversary and have competitive ratios that approach $e/(e-1)$. This compares with the ratio of 3/2 that can be achieved on an equilateral triangle.

Key Words. On-line algorithms, Competitive analysis, Randomized algorithms.

1. Motivation and Results. The amortized analysis of on-line algorithms for processing sequences of tasks in dynamic systems has been a subject of great interest in recent years. The approach taken is to compare the performance of a strategy that operates with no knowledge of the future with that of a clairvoyant strategy that has complete knowledge of the future and operates optimally given that information. A large number of problems have been studied from this point of view, see [1]–[8] and [10]–[20].

A randomized algorithm can be defined as a probability distribution on a set of deterministic algorithms. Let σ be an input sequence to a randomized algorithm A, and let $C_A(\sigma)$ be a random variable denoting A's cost on σ. We use the symbol $\mathbf{E}$ to denote an expected value over a probability distribution, so $\mathbf{E}C_A(\sigma)$ is the expected value of A's cost on sequence σ.

The randomized algorithms we present are *optimally competitive against an oblivious adversary*. Let $C_{opt}(\sigma)$ be the cost incurred by an optimal off-line algorithm in processing σ. An on-line algorithm A is said to be *c-competitive against an*

[1] DEC Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, USA.
[2] Department of Mathematics and Computer Science, Amherst College, Amherst, MA 01002, USA.
Supported in part by the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS), an NSF Science and Technology Center funded under NSF Contract STC-88-09648 and supported by the New Jersey Commission on Science and Technology.

Received August 7, 1990; revised September 11, 1991. Communicated by Prabhakar Raghavan.

oblivious adversary if there is a constant a such that, for any fixed input sequence σ,

$$\mathbf{E}C_A(\sigma) \leq c \cdot C_{opt}(\sigma) + a.$$

The constant c is known as the *competitive factor.* Algorithm A is called *optimally c-competitive* (or just *optimally competitive*) if it is c-competitive and if no algorithm is competitive within a factor less than c.

Sometimes competitive algorithms are considered for a problem that is really a class of problem instances. For example, the two-server problem (discussed in Section 5) depends on an underlying metric space M. An algorithm is c-competitive for such a problem if it is c-competitive for each instance. The additive constant a can depend on the instance.

The notion of an adversary arises in the definition of a competitive algorithm because we seek an algorithm that has a good performance ratio, even when handling a request sequence generated by a cruel adversary. An oblivious adversary is limited in power because it is oblivious to the random choices made by our algorithm. That is, once the request sequence is chosen, it cannot be modified based on the actions of the randomized algorithm.

Raghavan and Snir [19] defined two other adversaries that are more powerful and can adapt to the actions of the randomized algorithm. An *adaptive off-line adversary* can choose each input request depending on the choices made by the on-line algorithm in servicing the previous requests. Suppose the adversary selects an algorithm S to generate requests. The request sequence $\sigma(S)$ that is generated by S depends on the actions of the on-line algorithm, so it is a random variable. The length of the sequence generated by S must be bounded by some constant $f(S)$ regardless of the random choices. An on-line algorithm A is c-competitive against an adaptive off-line adversary if there is a constant a such that, for all S,

$$\mathbf{E}C_A(\sigma(S)) \leq c \cdot \mathbf{E}C_{opt}(\sigma(S)) + a.$$

An *adaptive on-line adversary* is similar, but it is limited in power because it must itself respond to the requests on-line. An on-line algorithm A is c-competitive against an adaptive on-line adversary if there is a constant a such that, for all S and on-line algorithms B,

$$\mathbf{E}C_A(\sigma(S)) \leq c \cdot \mathbf{E}C_B(\sigma(S)) + a.$$

For each adaptive adversary, a c-competitive algorithm is called *optimally c-competitive* (or just *optimally competitive*) if no algorithm achieves a competitive factor less than c against the same adversary.

For deterministic algorithms, discussion of adversaries is irrelevant because the weakest adversary (oblivious) can simulate the strongest one (adaptive off-line). Furthermore, Ben-David *et al.* [1] have shown that randomization adds no power against an adaptive off-line adversary. A central question in on-line computation concerns the extent to which randomization can help against a weaker adversary.

Two dramatic results have been obtained showing the power of randomization against an oblivious adversary. The first, due to Borodin *et al.* [4] shows that for an n-state metrical task system where all state transition costs are equal the competitive factor can be improved from $O(n)$ against an adaptive off-line adversary to $2H_n$ against an oblivious adversary. ($H_n = \sum_{i=1}^{n}(1/n) \in O(\log n)$.) A similar result due to Fiat *et al.* [10] shows that, for a paging problem with memory size k, the competitive factor can be improved from k (against either adaptive adversary [19], [20]) to $2H_k$ (against an oblivious adversary). They also show that no paging algorithm can be competitive within a factor less than H_k against an oblivious adversary. McGeoch and Sleator [18] later obtained an algorithm that achieves this bound.

In this paper we present new randomized algorithms for the snoopy caching and spin-block problems. For both of these problems, it is not possible to construct an on-line algorithm with a competitive ratio better than 2 against an adaptive on-line adversary. Our randomized algorithms are optimally competitive against an oblivious adversary, with competitive ratios that approach $e/(e-1) \approx 1.58$.

We also consider the question of how well a deterministic algorithm can perform if the input sequence is generated according to some unknown but time-independent probability distribution. This question was motivated by the observation that in traces obtained for programs running on snoopy caching multiprocessor systems [9], different programs exhibit vastly different input characteristics. In particular, the interprogram input variability was extremely high, while the intraprogram input variability was extremely low. This suggests that an algorithm which adapts to the observed input statistics can potentially converge to near-optimal behavior.

We show that adaptive deterministic algorithms of this type for snoopy caching and for the spin-block problem achieve competitive ratios which at worst approach $e/(e-1)$ if the input sequence is generated according to a fixed probability distribution. We further present practical and simple versions of the adaptive algorithms for these problems that are also 3-competitive against all adversaries.

Finally, we present upper- and lower-bound results for the two-server problem. It is known that for the two-server problem on an equilateral triangle, there is an optimally 3/2-competitive algorithm against an oblivious adversary. We show that such a competitive ratio cannot be achieved on all triangles. In particular, for the two-server problem on a class of isosceles triangles, there is a lower bound on the competitive ratio greater than 1.5 and tending to $e/(e-1)$. We also show a lower bound greater than 1.54 on the competitive ratio for the 2-server problem on the 3–4–5 triangle. We present algorithms that achieve these bounds, the first algorithms that are optimally competitive against an oblivious adversary on a nonuniform server problem.

2. Phase-Based Algorithms.

Each of the problems discussed in this paper can be represented as a finite *metrical task system* [4] with an initial state, a finite collection of other states, a transition matrix of nonnegative costs for moving from

one state to another, and a finite collection of possible tasks (with nonnegative costs that depend on the current state). The triangle inequality holds on the state transition costs. An algorithm must decide how to move the system from state to state in response to a sequence of requests, where each request is one of the tasks. At the moment a request arrives, the algorithm must choose a state for handling the task. The algorithm's cost on the request equals the cost (if any) of moving to the chosen state, plus the cost of performing the task in the chosen state. An on-line algorithm must choose the state without knowledge of later requests.

To simplify discussion we permit an algorithm to change state "immediately before," "at the moment of," or "immediately after" a request. Knowledge of a request cannot be used until the moment of the request. The costs of any transitions immediately before or after the request are simply added to the cost of the request. We use transitions immediately before and after requests in our analysis, but they cannot actually help an algorithm, because any transition can be deferred without extra cost to the moment of the next request. When we discuss the state of the system "after" a request, we mean its state after all the transitions have been made, including those immediately after the request.

One of the problems we study, the snoopy caching problem, sometimes uses *look-ahead-zero* requests. These are similar to the usual *look-ahead-one* requests described above, but no state change is permitted at the moment of a request. All state changes occur either immediately before or immediately after a request. The system must handle the request in whatever state it happens to be in at the moment of the request. The results in this section apply to task systems with either or both of these kinds of requests.

For a finite metrical task system, there is an optimal off-line algorithm based on dynamic programming; we call this algorithm *opt*. An on-line algorithm can perform the dynamic programming and determine the optimal off-line cost for reaching a given state after processing the requests so far. Sometimes it is possible for an on-line algorithm to "know" *opt*'s current state. We can assume that *opt* is in state s_0 after some request if its cost in each state s is at least its cost in s_0 plus the cost of a move from s_0 to s. We can say that *opt is known to be in state* s_0 whenever this situation occurs. (If, and only if, there are zero-cost transitions in both directions between two states, it is possible for *opt* to be known to be in more than one state. In this case we say that *opt* is known to be in whichever of these states comes first in some canonical order.)

If *opt*'s state is known and if an algorithm A has positive probability of being in some different state, then a cruel adversary can make requests which increase A's expected cost without affecting the optimal cost. This idea leads to the following theorem.

THEOREM 1. *Suppose A is an α-competitive algorithm for a finite metrical task system. Suppose that for each state s there is a finite request sequence ρ which has:*

(1) *zero cost in state s, and*
(2) *positive cost for algorithms that never enter s.*

There must be an α-competitive algorithm A' that is in the same state as opt after each request for which opt's state is known.

PROOF. Let ε be the smallest positive cost in the state transition matrix or in the cost vector of any request. Let E be the largest state transition cost. To simplify the proof we assume that (except on the first request) A makes no move "immediately before" a request. Such moves can be combined with the move "immediately after" the previous request.

Algorithm A' handles each request sequence σ by simulating A on a longer sequence σ' which includes extra requests that are free for *opt*. This means that $C_{opt}(\sigma') = C_{opt}(\sigma)$. Algorithm A' chooses its moves based on the moves made by A. The cost of A' on σ is no greater than the cost of A on σ'. Summing over A's probability distribution, we get $\mathbf{E}C_{A'}(\sigma) \leq \mathbf{E}C_A(\sigma')$. Therefore, if A is α-competitive there is some a such that

$$\mathbf{E}C_{A'}(\sigma) \leq \mathbf{E}C_A(\sigma') \leq \alpha \cdot C_{opt}(\sigma') + a = \alpha \cdot C_{opt}(\sigma) + a,$$

and thus A' is α-competitive.

It remains to describe algorithm A'. It passes each new request to a simulator for A, and works in the following way. If *opt*'s state is not known after the request, A' simply mimics A's response. If *opt* is known to be in state s after the request, A' mimics A until immediately after the request. At that moment, it moves to s and gives A a long sequence of requests, cycling $2E/\varepsilon$ times through the finite sequence ρ, where ρ has zero cost in s and positive cost for algorithms that never enter s. If A ends up in some state $s' \neq s$, A' stays in s, but moves to s' immediately before the next request.

This procedure ensures that A' is in the same state as *opt* after each request for which *opt*'s state is known. It also ensures that A' is in the same state as A at the moment each request arrives. The cost of A' on each request is at most the cost that A incurs for its requests. This is clear if A ends in state s or if *opt*'s state is not known. If A enters s and later switches to s', then its cost is at least enough to cover A's move to s as well as its later move to s'. If A never enters s, then its cost is high: at least ε on each pass through ρ. There are $2E/\varepsilon$ passes, yielding total cost $2E$. This is enough to cover the cost of A''s moves to s and s'. □

This theorem means that in order to prove upper and lower bounds on the competitive factor for a problem, it suffices to consider only algorithms that are in the same state as *opt* whenever *opt*'s state is known. If some algorithm achieves a given competitive factor, then it can be modified to meet this restriction and achieve the same factor. On the other hand, a lower bound for algorithms restricted in this way immediately implies a lower bound for all algorithms.

Given that upper and lower bounds for our problems can be obtained by considering a restricted class of algorithms, it is also possible to restrict the kinds of request sequences that need to be considered. Suppose that a request has expected cost zero for all of the restricted algorithms and does not cause any of

them to change state. Such a request is essentially uninteresting. If we forbid such requests, it cannot increase the competitive factor of any of the algorithms. So any lower bound obtained using restricted requests will apply in general. Furthermore, if forbidding uninteresting requests can actually improve an algorithm's competitive factor, then a better algorithm can be obtained in giving it a preprocessor that filters out uninteresting requests. Therefore any upper bound on a problem's competitive factor that is found using restricted requests will apply in general.

Our results use a key property which holds for each of our problems: every request sequence (subject to the restriction above) can be partitioned into *phases*. Each phase begins and ends with *opt*'s state being known, and restrictions (1) and (2) from Theorem 1 must hold for *opt*'s starting and ending states. The lengths of all phases must be uniformly bounded by a single constant, and *opt*'s costs on all phases must be positive and uniformly bounded by a constant. (The final phase may be incomplete, but it must still obey the constant bounds on length and on *opt*'s cost.) Each phase can be characterized by *opt*'s starting and ending states and by the sequence of requests. It follows that there is only a finite set of possible phases. (For a given problem, the ability to obtain a finite list of bounded-length phases depends critically on being able to ignore requests that are uninteresting, in the sense described above.)

Most of the on-line algorithms we discuss in this paper are *phase-based*, meaning that:

(1) They are in the same state as *opt* at the end of each phase.
(2) Their actions do not depend on requests that preceded the current phase.

If A is a phase-based algorithm, we write $C_A(\rho)$ to denote its cost on phase ρ. As before, $C_A(\rho)$ is a random variable if A is randomized. This extends the previous definition of cost, because the original definition depended on the task system's initial state. When discussing an algorithm's cost on a phase, we mean the algorithm's cost when starting in whatever state *opt* used at the beginning of the phase.

We can calculate an upper bound on the competitive factor of a phase-based algorithm by minimizing a linear program. Let variable α represent this bound (which we call the *LP bound*), and let one variable Φ_s represent the *potential* of each state s. The linear program uses one constraint for each possible phase, and the objective function is "minimize α." Suppose that ρ is a phase that takes *opt* from state s to s'. We calculate $\mathbf{E}C_A(\rho)$ and then use the constraint

$$\mathbf{E}C_A(\rho) \leq \alpha \cdot C_{opt}(\rho) + \Phi_s - \Phi_{s'}.$$

THEOREM 2. *If A is a phase-based algorithm and if α is its LP bound, then A is α-competitive.*

PROOF. Suppose σ is a request sequence. Let σ' be the sequence obtained by deleting any incomplete phase at the end of σ. Recall that *opt*'s state is known after each phase of σ and that A is (with probability 1) in the same state. Let Φ_0 be the potential (obtained from the minimization) of the initial state, let p be the

number of phases, let ρ_i be the ith phase, and let Φ_i be the potential after the ith phase. Then

$$\begin{aligned}\mathbf{E}C_A(\sigma') &= \sum_{i=1}^{p} \mathbf{E}C_A(\rho_i) \\ &\leq \sum_{i=1}^{p} (\alpha \cdot C_{opt}(\rho_i) + \Phi_{i-1} - \Phi_i) \\ &= \alpha \cdot C_{opt}(\sigma') + \Phi_0 - \Phi_p.\end{aligned}$$

If there is an incomplete phase at the end of the request sequence, it contributes at most a constant to A's expected cost. This holds because the incomplete phase is a prefix of some complete phase, A's expected cost on each complete phase is within a factor α of *opt*'s cost, and *opt*'s cost in a phase is bounded by a constant. □

THEOREM 3. *Suppose that a task system has the property that every request sequence can be partitioned into phases. If α is the infimum of the LP bounds for phase-based algorithms, then no algorithm is competitive within a factor less than α.*

PROOF. Suppose some algorithm A is competitive within a factor $\alpha' < \alpha$. We show how to build a phase-based algorithm A' with an LP bound less than α, which is a contradiction. We assume (by Theorem 1) that A is in the same state as *opt* whenever *opt*'s state is known.

Since A is α'-competitive, there is some constant a such that, for all σ, $\mathbf{E}C_A(\sigma) \leq \alpha' \cdot C_{opt}(\sigma) - a$. Now consider those request sequences after which *opt* is known to be in a state s, and define Φ_s to be the infimum of $\alpha' \cdot C_{opt}(\sigma) - \mathbf{E}C_A(\sigma)$ for such strings. (This value is clearly at least a.) Let σ_s be a sequence of requests after which *opt* is known to be in state s and for which

$$\Phi_s + (\alpha - \alpha')\varepsilon > \alpha' \cdot C_{opt}(\sigma_s) - \mathbf{E}C_A(\sigma_s),$$

where ε is the minimum positive cost in the task system. Such a sequence must exist, for otherwise Φ_s could be increased to $\Phi_s + (\alpha - \alpha')\varepsilon$.

Algorithm A' can now be built in the following way. In a phase beginning in state s, let A' mimic the actions of A, after "warming up" A by giving it σ_s. After processing σ_s, A is in state s with probability 1. If a phase ρ takes *opt* from s to s', we know that A will be in state s' at the end of the phase. Since A' mimics A throughout the phase, it will also be in state s' at the end of the phase. The expected cost of A' on the phase is

$$\mathbf{E}C_{A'}(\rho) = \mathbf{E}C_A(\sigma_s\rho) - \mathbf{E}C_A(\sigma_s).$$

Now by the definition of $\Phi_{s'}$,

$$\mathbf{E}C_A(\sigma_s\rho) \leq \alpha' \cdot C_{opt}(\sigma_s\rho) - \Phi_{s'},$$

and by the choice of σ_s,

$$\mathbf{E}C_A(\sigma_s) > \alpha' \cdot C_{opt}(\sigma_s) - \Phi_s - (\alpha - \alpha')\varepsilon.$$

Therefore

$$\mathbf{E}C_{A'}(\rho) = \mathbf{E}C_A(\sigma_s\rho) - \mathbf{E}C_A(\sigma_s) < \alpha' \cdot C_{opt}(\rho) + \Phi_s - \Phi_{s'} + (\alpha - \alpha')\varepsilon.$$

Rearranging yields

$$\mathbf{E}C_{A'}(\rho) < \alpha \cdot C_{opt}(\rho) + (\alpha' - \alpha)(C_{opt}(\rho) - \varepsilon) + \Phi_s - \Phi_{s'}.$$

Now recall that *opt* has positive cost in every phase. The minimum positive cost in the task system is ε, so therefore $C_{opt}(\rho) \geq \varepsilon$. Furthermore, the quantity $(\alpha' - \alpha)$ is negative, so it follows that

$$\mathbf{E}C_{A'}(\rho) < \alpha \cdot C_{opt}(\rho) + \Phi_s - \Phi_{s'}.$$

Let $\alpha_\rho < \alpha$ be the value for which

$$\mathbf{E}C_{A'}(\rho) = \alpha_\rho \cdot C_{opt}(\rho) + \Phi_s - \Phi_{s'}.$$

Let $\hat{\alpha}$ be the maximum over all phases ρ of α_ρ. Recall that the number of phases is finite, so the maximum does exist. Since each α_ρ is less than α, so is $\hat{\alpha}$. It is clear that, for each phase ρ,

$$\mathbf{E}C_{A'}(\rho) \leq \hat{\alpha} \cdot C_{opt}(\rho) + \Phi_s - \Phi_{s'}.$$

This is precisely the constraint needed for phase ρ in the procedure for computing the LP bound for A'. Because this inequality holds for each phase ρ, using $\hat{\alpha}$ and the Φ values given above, the LP bound for A' must be at most $\hat{\alpha}$. This contradicts the claim that α is the infimum of LP bounds for phase-based algorithms. □

3. Snoopy Caching

3.1. The Block Snoopy-Caching Model. A snoopy-caching multiprocessor system is a system with a set of processors, each with its own memory cache, where the caches are all connected over a common bus. In this work we use the *block snoopy-caching model* described by Karlin *et al.* [14]. In this model there is a single

address space used by all of the processors. Each location in the space is called a *variable*. The memory space is partitioned into uniform *blocks* containing $p - 1$ variables, and the cache belonging to each processor is large enough to hold the entire address space. A block is *valid* at a particular time in a given cache if the cache's copy of the block is marked so that it will be kept up-to-date whenever any processor writes to a variable in the block; otherwise it is *invalid*. A block is *private* at some time if it is valid in only one cache; otherwise it is *shared* and valid in all caches containing it. We consider algorithms that decide which blocks should be valid in which caches. The goal of these algorithms is to minimize the number of bus cycles used.

All of a processor's memory requests are serviced through its cache. The cost of reading a variable depends on whether or not the block containing that variable is valid in the requesting cache. If it is, then the read is executed at no cost. If the block is invalid, then the cache must send out a request for the appropriate block. This block is then broadcast on the bus by some other cache. The cost of this read is p bus cycles (one cycle for the request and one cycle for each variable in the block). At the time a block is broadcast on the bus, all caches can make a copy of it; this is the sense in which the caches are "snoopy."

We assume that every variable into which a processor writes is in a block that is valid in the processor's cache. If the block is private, the write is done locally and no bus cycles are used to communicate the change to other memories. If the block is shared, then one bus cycle must be used to announce the variable and its new value. We assume that every write is preceded by a read to the same block. If the block is already valid in the processor's cache, the read has no cost. Otherwise, the read is required before the write can occur.

A *snoopy-caching algorithm* for a single block B must decide how to handle requests in the following task system. There is one state for each processor, denoting that B is private to that processor's cache, and one state for which B is shared between all caches. (If a block is shared by more than one cache, it may as well be shared by them all.) A read from processor i is a look-ahead-one request that has zero cost if B is shared or private to i's cache. Otherwise it has infinite cost, forcing a state change. A write from processor i is a look-ahead-zero request with cost zero in i's private state, cost one in the shared state, and infinite cost otherwise. There is a state-transition cost p for going from a private state to any other state, and a cost zero for going from the shared state to any private state.

In the proof that follows it is shown that there is an optimally competitive snoopy-caching algorithm where all state transitions occur on requests with nonzero cost, that is, on requests that cause bus activity. This is fortunate, because it permits the algorithm to be implemented on a distributed basis. If all caches hear the same bus traffic, if they all use the same algorithm, and if they all make pseudorandom choices in the same way, then they can make consistent choices about how memory blocks are shared. Each cache can copy all of the blocks that are transferred and record updates to variables in any shared blocks. After each broadcast, the caches may simultaneously decide to end sharing of the corresponding block, leaving a valid copy only in the cache that did the last read or write.

Karlin*et al.* [14] discovered an optimally 2-competitive deterministic algorithm for block snoopy caching. We now show that it is possible to improve this ratio against an oblivious adversary by randomizing.

3.2. Randomized Algorithms for Snoopy Caching. For any constant $p > 0$, define an associated constant e_p as follows:

$$e_p = \left(1 + \frac{1}{p}\right)^p.$$

As p approaches infinity, this value approaches e.

THEOREM 4. *Consider a block snoopy-caching multiprocessor system with block size $p - 1$. No algorithm is competitive against an oblivious adversary within a factor less than $e_p/(e_p - 1)$. There is a randomized algorithm that achieves this competitive factor.*

PROOF. The problem of handling a sequence of read and write requests for a collection of memory blocks can be divided into separate problems for each of the blocks. Each cache has enough space to hold any or all blocks, so the choices made for different blocks do not interact in any way. We therefore concentrate on the request sequence for a single block B. The system can initially have B shared or private to one of the caches.

We show that no phase-based on-line algorithm for this problem has an LP bound less than $e_p/(e_p - 1)$. The lower bound is then obtained by applying Theorem 3. We then show a randomized algorithm that achieves this bound, proving the theorem.

Every request sequence for a snoopy caching problem can be partitioned into *write runs*, maximal contiguous blocks of reads and writes from a single processor. (They are called write runs because it is the writes that are significant in determining an algorithm's cost. Recall, however, that every write is preceded by a read for the same block.) The phases used in our proof correspond to the write runs. In general, each phase begins after the initial read of a write run and ends after the initial read of the next write run. To handle a read request from a new processor, the optimal algorithm *opt* must be in the shared state or in the state where the block is private to the new processor's cache. Both have the same cost, and there is no advantage to making the block private to one processor at this time, so *opt* is "known to be in the shared state," in the sense described earlier.

If the system begins with block B being private to one processor, there is an initial phase that ends after the first read from some other processor. With this exception, every phase begins and ends with *opt* in the shared state. We ignore this initial phase in the lower bound proof, and we later deal with it easily in the upper bound proof.

The cost of optimal algorithm *opt* can be easily computed. Recall that every write in a snoopy-caching request sequence is preceded by a read from the same

processor. Suppose that *opt* begins a phase with block B in the shared state. Its behavior during the phase depends on the number of writes in the phase. If there are $k < p$ writes, *opt* will continue sharing throughout the phase. Each write will have cost one, for a total cost of k for the phase. If $k \geq p$, *opt* may as well make B private to one cache immediately before the first write of the phase. It incurs no cost until some other processor accesses B, at which time it incurs cost p to rebroadcast B across the bus. In either case it is clear that *opt* is in the shared state after the first read from a new processor.

We can assume (by Theorem 1) that any on-line algorithm A will be in the same state as *opt* after any request where *opt*'s state is known. This means that A and *opt* will be in the same state at the beginning and end of a phase. It also means that A must make block B private to the active cache after the first p writes of a phase. In analyzing A's competitive factor we can disregard any writes after the first p of a phase, because they must be free from A. We can also disregard any reads after the initial read of the phase; they are free for A regardless of whether or not the caches are sharing the block. Any algorithm can be modified to ignore such reads, without increasing its competitive factor. Therefore we can restrict our attention to phases which consist of a read followed by up to p writes from the same processor.

If A is phase-based, then it must make its decisions on the basis of the number of writes so far in current phase. For each k from 1 to p, there must be a fixed probability π_k that A has made the block private to the active processor before the kth write. The probabilities are cumulative, so $0 \leq \pi_1 \leq \cdots \leq \pi_p \leq \pi_{p+1} = 1$, where $\pi_{p+1} = 1$ is the probability that the block is private after p writes.

Let σ_k denote a phase with k writes. As noted above, we can assume that no phase has more than p writes. For each k from 0 to p, A's expected cost is

$$\mathbf{E}C_A(\sigma_k) = \pi_{k+1}p + \sum_{i=1}^{k} (1 - \pi_i).$$

In this equation $(1 - \pi_i)$ is the probability that the ith write requires a variable to be updated on the bus, and π_{k+1} is the probability that a block must be transferred to handle the read which begins the next phase. Recall that the cost is 1 to broadcast a variable and p to transfer an entire block. For each k from 0 to p, *opt*'s cost is

$$C_{opt}(\sigma_k) = k.$$

If A has an LP bound of α, it must be that, for each k from 0 to p,

$$\mathbf{E}C_A(\sigma_k) \leq \alpha \cdot k.$$

To establish a lower bound on the LP bound of any phase-based algorithm, we permit the π_k to be free variables and we minimize α subject to these constraints.

Variable π_1 has a positive coefficient on the left-hand side of one of the inequalities and a negative coefficient on the left-hand side of all of the others. The positive coefficient appears in $\pi_1 p \leq 0$. Suppose that the minimum value of α was obtained with $\pi_1 < 0$. We could raise π_1 to 0, which would only decrease the left-hand side of the other bounds. Thus we can replace the bound $\pi_1 p \leq 0$ with $\pi_1 p = 0$ without affecting the minimum α. Now consider π_2. It has a positive coefficient only in $\pi_2 p + 1 - \pi_1 \leq \alpha$. Suppose that the minimum value of α was obtained with $\pi_2 p + 1 - \pi_1 < \alpha$. We could raise π_2 to make the bound tight, which would only decrease the left-hand side of the other bounds. Thus we can replace the bound $\pi_2 p + 1 - \pi_1 \leq \alpha$ with $\pi_2 p + 1 - \pi_1 = \alpha$ without affecting the minimum α. Continuing in this way, each other π_k appears with a positive coefficient in one inequality and a zero or negative coefficient in the rest. So, for each k from 0 to p, we can replace the inequality $\pi_{k+1} p + \sum_{i=1}^{k}(1 - \pi_i) \leq \alpha k$ by the equation $\pi_{k+1} p + \sum_{i=1}^{k}(1 - \pi_i) = \alpha k$. It follows that $\pi_1 = 0$, and, for each k from 1 to p,

$$\begin{aligned}
\alpha k &= \pi_{k+1} p + \sum_{i=1}^{k} (1 - \pi_i) \\
&= \pi_{k+1} p + (1 - \pi_k) + \sum_{i=1}^{k-1} (1 - \pi_i) \\
&= \pi_{k+1} p + (1 - \pi_k) + (\alpha(k-1) - \pi_k p) \\
&= \pi_{k+1} p + 1 - \pi_k(p+1) + \alpha(k-1).
\end{aligned}$$

Therefore, for each k from 1 to p,

$$\pi_{k+1} = \frac{\alpha - 1 + \pi_k(p+1)}{p}.$$

It is then easy to verify that, for each k from 1 to $p+1$,

$$\pi_k = (\alpha - 1)\left(\left(\frac{p+1}{p}\right)^{k-1} - 1\right).$$

Since $\pi_{p+1} = 1$, it follows that

$$\alpha = \frac{1}{((p+1)/p)^p - 1} + 1 = \frac{e_p}{e_p - 1}.$$

This is a lower bound on the LP bound of any phase-based algorithm. By Theorem 3, this is a lower bound on the competitive ratio of any algorithm.

A phase-based algorithm that achieves this competitive factor can be obtained by simply using the π_k obtained in the minimization. It is easy to show that

$0 \leq \pi_1 \leq \cdots \leq \pi_p \leq 1$, so an algorithm can be constructed where the block has probability π_k of being private by the time of the kth write. This ensures that the algorithm's expected cost is within factor α of *opt* on each phase. (If the system begins with the block private to one processor, the algorithm can ignore all requests until the first read from some other processor, incurring cost p for the initial phase.) □

It is useful to note that randomized algorithms can only achieve a competitive factor of 2 against an adaptive on-line adversary. Thus the deterministic algorithm of Karlin *et al.* [14] is also optimally competitive against this adversary.

THEOREM 5. *There is no c-competitive algorithm for the snoopy-caching problem for $c < 2$ against an adaptive on-line adversary.*

PROOF. Suppose that A is a randomized on-line algorithm. An on-line adversary can build a request sequence that refers to a single cache block. The sequence is based on write runs consisting of reads and writes from a single processor. Whenever A decides to make the block private to one cache, the adversary begins a write run for some other processor. As in the proof against an oblivious adversary, a new phase begins immediately after the first read of the new write run.

In a given phase let W be the expected number of writes before A makes the block private. If $W \leq p$, the adversary's algorithm remains in the shared state throughout the phase. A's expected cost for the phase is $W + p \geq 2W$, while the adversary's cost is W. If $W > p$, the adversary makes the block private immediately after the first read of the phase. A's expected cost is $W + p > 2p$, while the adversary's cost is p.

The on-line adversary can repeat this process for as many phases as desired. Because A's expected cost in each phase is at least twice the on-line adversary's cost, A cannot be competitive within a factor less than 2. □

3.3. Randomized Algorithms for Limited Block Snoopy Caching. In this section we extend the results for block snoopy caching to the more realistic model of finite direct-mapped caches. In this model each processor has a finite cache, and there is also a main memory of infinite size. Each cache consists of a number of *cache lines*, slots that may contain blocks. A *direct-mapped* cache i uses a hash function $h_i(B)$ to determine the unique cache line in which a block B will reside. If $h_i(B) = h_i(B')$, then cache i can contain at most one of the blocks B and B' at any time. Since different blocks can occupy the same cache line, a block that is read into a line may displace a block which was written to privately. In this case the line is *dirty* and must be *written back* to main memory at a cost of p. The assumption that the main memory is infinite means that it is large enough to hold copies of any blocks that are not in some cache.

In the *limited block snoopy-caching* model, a cache works in a similar way to the block snoopy-caching model described earlier. The only difference concerns the extent to which caches "snoop" and make copies of blocks being sent over

the bus between other caches. In the limited model a cache makes a copy of a block B only if:

(1) No block is currently occupying the corresponding line.
(2) B was the last block to occupy its line.

We can extend Theorem 4 to cover the limited block snoopy-caching model, assuming that all algorithms use a given hash function h_i on each cache i.

THEOREM 6. *Consider a limited block snoopy-caching multiprocessor system with block size $p - 1$. No algorithm is competitive against an oblivious adversary within a factor less than $e_p/(e_p - 1)$. There is a randomized algorithm that achieves this competitive factor.*

PROOF. The lower bound follows easily from the lower bound of Theorem 4, because that bound was proven by considering requests for a single block. The two models are identical in this case.

To prove the upper bound, we break an input sequence σ into write runs consisting of reads and writes for a single block B from a single processor. Each write run for B is a maximal sequence of requests for B that ends when one of two things happens:

(1) There is a read for B from some other processor.
(2) The processor that is using B issues a read for a block that shares the same cache line with B.

Write runs may be interleaved, that is, there may be simultaneous write runs in progress for distinct blocks.

The upper bound is achieved by an algorithm A which uses the number of writes in B's current write run in order to decide when to make B private. Algorithm A uses the probabilities obtained in Theorem 4.

We show how σ can be transformed into a new sequence σ'. The optimal cost for the two sequences is the same under the limited block model and equals the optimal cost under the block snoopy-caching model. A's expected cost in the limited block model is not decreased by the transformation, and A's expected cost on σ' is the same for both the block and limited block models. Using C to denote cost under the block model and $\hat{C}$ to denote cost under the limited block model, it follows that

$$\mathbf{E}\hat{C}_A(\sigma) \leq \mathbf{E}\hat{C}_A(\sigma') = \mathbf{E}C_A(\sigma') \leq \alpha \cdot C_{opt}(\sigma') + a = \alpha \cdot \hat{C}_{opt}(\sigma) + a,$$

where α and a are the constants from the block snoopy-caching model. This proves the theorem.

Sequence σ can have different costs under the two models for only two reasons. The sequence may incur extra costs under the limited block model for:

(1) Writing a dirty block back to the main memory.
(2) Reading from a block that was previously bumped out.

The transformation ensures that these situations do not arise with σ', so any algorithm will have the same cost on σ' under both models. It remains only to show that *opt*'s cost under the limited block model is the same for both σ and σ' and that A's cost under the limited block model is not decreased by the transformation.

Situation (1) occurs when a write run for block B terminates in σ because the active processor reads some block which uses the same cache line. To eliminate this problem, σ' is built by using the same requests as σ, with a read for B from the main memory inserted immediately before the colliding read. (The main memory can be thought of as a processor with a large cache in which each block has a separate line.) Adding this request does not change the cost of either A or the optimal algorithm. In each case, if B was shared at the time of the read from main memory, the main memory already contains a valid copy and the request is free. If B was private to the active cache, the request would have cost p, replacing the cost in σ of writing back the block at the time of the collision.

Situation (2) occurs when a write run for block B terminates in σ because some other processor issues a read for B and places it in a cache line that was dedicated to some other block. Under the limited block model any algorithm will incur cost p for the read. Under the block model each block has its own line, and the read would be free if B is shared at the time of the request. Whenever situation (2) occurs, we can modify σ by adding p extra writes to the write run. The optimal algorithm's cost under the limited block model is not affected by the transformation, because the optimal limited block strategy in both σ and σ' is to make B private throughout the write run. Furthermore, the expected cost of A is not decreased by the transformation. Recall that A's expected cost on a sequence equals the expected number of writes for which B is shared, plus p if a broadcast is done at the end of the write run. Adding additional writes to the write run does not decrease the expected number of writes for which B is shared, and A must pay p for the broadcast in both σ and σ'. □

3.4. Adaptive Algorithms. Traces obtained for programs running on snoopy-caching multiprocessor systems show that different programs exhibit vastly different write-run characteristics. Specifically, the programs analyzed in [9] have the property that either write runs are very short (variables are actively being shared) or they are very long (there is very little sharing going on). In the first case an algorithm where blocks are always shared performs best; in the second case an algorithm where blocks are always private to the active cache performs best.

Motivated by these observations, we studied the following problem. Suppose that the request sequence for a given block consists of write runs chosen independently from some unknown probability distribution $\mathscr{P}$. (Each block may have a different distribution.) How well can a deterministic on-line algorithm do? The following theorem shows that an adaptive algorithm is competitive within a small factor.

THEOREM 7. *Consider the block snoopy-caching problem with block size* $p - 1$, *where request sequences have write runs chosen independently according to a*

probability distribution $\mathcal{P}$. There is a deterministic algorithm that is c-competitive for all $c > e_p/(e_p - 1)$.

PROOF. We first describe a deterministic $e_p/(e_p - 1)$-competitive algorithm A^* for request sequences generated according to $\mathcal{P}$, and then we construct an adaptive algorithm $\hat{A}$ with a competitive factor that approaches that factor of A^*. The adaptive algorithm works without advance knowledge of the distribution.

Let A_i be a deterministic algorithm that makes a block private after i consecutive writes from the active processor. The randomized algorithm A described in Theorem 4 uses algorithm A_i in a given write run with probability we call p_i. (This value equals $\pi_{i+1} - \pi_i$ if $1 < i \leq p$ and equals zero otherwise.) The expected cost of A on a write run σ equals $\sum_i p_i C_{A_i}(\sigma)$. Because this expected cost is within a factor $e_p/(e_p - 1)$ of *opt*'s cost, we have

$$\sum_i p_i C_{A_i}(\sigma) \leq \frac{e_p}{e_p - 1} C_{opt}(\sigma).$$

Averaging over write runs $\sigma(\mathcal{P})$ generated according to $\mathcal{P}$ yields

$$\sum_i p_i \mathbf{E} C_{A_i}(\sigma(\mathcal{P})) \leq \frac{e_p}{e_p - 1} \mathbf{E} C_{opt}(\sigma(\mathcal{P})).$$

Now, there must be some A_i such that

$$\mathbf{E} C_{A_i}(\sigma(\mathcal{P})) \leq \frac{e_p}{e_p - 1} \mathbf{E} C_{opt}(\sigma(\mathcal{P})),$$

because otherwise the summation would exceed the required bound. Let A^* be an algorithm A_i that minimizes $\mathbf{E} C_{A_i}(\sigma(\mathcal{P}))$ over all i. Clearly, this algorithm is $e_p/(e_p - 1)$-competitive.

Since distribution $\mathcal{P}$ may not be known in advance, an obvious approach is for an on-line algorithm to collect statistics on write-run lengths. It can choose an algorithm A_i that minimizes the expected cost on write runs generated according to the observed distribution and then use that algorithm in handling subsequent write runs. Let $\hat{A}$ be an adaptive algorithm that works in this way. By the law of large numbers, the statistics will converge to their true values. This means that the expected cost of $\hat{A}$ on a write run will approach the expected cost of A^*. Now suppose the expected cost of $\hat{A}$ on a write run falls within $(1 + \varepsilon)$ times the expected cost of A^*. Algorithm $\hat{A}$ is then competitive within ratio $(1 + \varepsilon)e_p/(e_p - 1)$ on all subsequent write runs. An additive constant depending on ε can cover $\hat{A}$'s expected cost up to that point. It follows that $\hat{A}$ is c-competitive for all $c > e_p/(e_p - 1)$. □

In practice, we feel that it should be sufficient to use the last few write runs as a sample. That is, if the last j write runs have lengths $i_1, i_2, \ldots, i_j$, then we

approximate $\mathscr{P}$ with the distribution $\mathscr{P}'$ consisting of sequences $\sigma_{i_1}, \sigma_{i_2}, \ldots, \sigma_{i_j}$, each with probability $1/j$. It is possible to show that the adaptive algorithm so obtained is still competitive.

THEOREM 8. *Let A' be the deterministic on-line algorithm that minimizes the expected cost for the distribution $\mathscr{P}'$ obtained by letting $j = 1$. Then A' is 3-competitive against all adversaries.*

PROOF. The algorithm obtained by adapting to the previous sample statistic is the following. Suppose that the last write run had length l. Then the distribution $\mathscr{P}'$ assumes that write runs have length l with probability 1. Consequently, the algorithm that minimizes the expected cost uses algorithm A_p on the next write run if $l \leq p$ and algorithm A_0 if $l > p$.

To see that this algorithm is 3-competitive, we simply observe that if A_p is used, then a ratio of 2 between on-line and off-line costs is achieved [14]. On the other hand, if A_0 is used, then the previous write run had length greater than p, and consequently the adversary paid at least p for that write run. Hence, we can charge our cost on the current write run to the adversary's cost on the previous write run, yielding an overall competitive ratio of 3. □

The larger the number of sample statistics used, the better the algorithm will perform. In any case, we believe the combination of the competitiveness of this algorithm with its adaptiveness to the potential underlying request distribution makes it eminently practical.

4. Spin–Block

4.1. The Problem. Consider a process that is waiting for a lock on a multiprocessor system where:

(a) There are other processes waiting for a processor.
(b) The process holding the lock is assigned to a processor.

There are two choices for the actions that may be taken: The process can *spin*, at a cost proportional to the length of time it spins, or it can *block*. The latter action has some large cost C reflecting the cost of restarting the process and restoring its state, usually referred to as the *context-switch* cost. Spinning may be useful, because the lock may become available immediately. On the other hand, spinning prevents the other processes that are waiting from getting a processor. The choice to spin or block can be solved optimally depending on how long it will be before the lock is freed, but this information is generally unavailable on-line. An on-line algorithm for the spin–block problem must decide how long a process should spin before it blocks.

The spin–block problem is a continuous version of the 2-cache, 1-block snoopy-caching problem. It is easy to construct a deterministic on-line algorithm

with cost at most twice that of the optimal off-line algorithm—just have the process spin for an amount of time equal to the cost of a context switch. As the next theorem shows, this is the optimal competitive factor, even for a randomized algorithm against an adaptive adversary.

THEOREM 9. *There is no c-competitive algorithm for the spin–block problem for* $c < 2$ *against an adaptive on-line adversary.*

PROOF. Suppose that A is a randomized on-line algorithm for determining when to block, and suppose that the on-line adversary always releases a lock as soon as A stops spinning. Whenever a process begins waiting for a lock, let S be A's expected spinning time before blocking. If $S \leq C$, the adversary's algorithm spins until the lock is available. A's expected cost is $S + C \geq 2S$, while the adversary's expected cost is S. If $S > C$, the adversary's algorithm blocks immediately. A's expected cost is $S + C > 2C$, while the adversary's cost is C. □

As the following theorem shows, our snoopy-caching results against an oblivious adversary can be extended to the spin–block problem.

THEOREM 10. *No algorithm for the spin–block problem is competitive against an oblivious adversary within a factor less than* $e/(e-1)$. *There is a randomized algorithm that achieves this competitive factor.*

PROOF. Suppose that an algorithm is restricted to spinning for a time equal to kC/p, where p is a fixed positive integer, k is any nonnegative integer, and C is the cost of blocking. Then the spin–block problem is identical to a 2-cache, 1-block snoopy-caching problem with page size $p-1$. The beginning of a processor wait for a lock is equivalent to a read from the inactive cache in the snoopy-caching problem. The passing of time C/p is equal to a write request. The decision to spin during the ith interval is equivalent to having the memory block shared at the moment of the ith write. For each time period, spinning carries cost C/p. The cost of blocking, C, corresponds to the cost a caching algorithm must pay to re-broadcast a private block.

The unrestricted spin–block problem is then equivalent to the snoopy-caching problem where p goes to infinity. Theorem 4 shows that as p approaches ∞, the snoopy-caching lower bound approaches $e/(e-1)$.

A randomized algorithm A achieving this competitive factor can be obtained by using the cumulative probability distribution

$$\pi(t) = \begin{cases} \dfrac{e^{t/C}-1}{e-1}, & 0 \leq t \leq C, \\ 1, & t > C, \end{cases}$$

where $\pi(t)$ denotes the probability that the algorithm blocks sometime before time t. A's expected cost on a lock σ_τ that is released at time τ is

$$\mathbf{E}C_A(\sigma_\tau) = \pi(\tau) \cdot C + \int_0^\tau (1 - \pi(t))\, dt.$$

The first part of the right-hand side denotes the expected cost associated with blocking. The second part denotes the expected cost of spinning, because $1 - \pi(t)$ is the probability that the process is spinning at time t. The cost of algorithm *opt* is

$$C_{opt}(\sigma_\tau) = \begin{cases} \tau, & 0 \le \tau \le C, \\ C, & \tau > C. \end{cases}$$

Substituting for $\pi(t)$ in the equation for $\mathbf{E}C_A$, it is straightforward to show that A is competitive within factor $e/(e-1)$.

(We derived the distribution $\pi(t)$ by noting that $\pi(t)$ should equal the value of π_k (from the proof of Theorem 4) as p goes to infinity, where $k = pt/C$.) □

The same results that hold for the adaptive setting of snoopy caching hold for the adaptive setting of the spin–block problem, as described in the following theorem.

THEOREM 11. *Consider the spin–block problem with lock-waiting times chosen independently according to a probability distribution $\mathcal{P}$. There is a deterministic algorithm that is c-competitive for all $c > e/(e-1)$.*

PROOF. The proof is similar that of Theorem 7. We first describe a lower bound C^* on the expected cost of certain algorithms for handling lock waits generated according to $\mathcal{P}$. This value is within $e/(e-1)$ times *opt*'s expected cost. We then describe an adaptive algorithm $\hat{A}$ for which the expected cost on a lock wait approaches C^*. This algorithm works without advance knowledge of the distribution.

Let A_t be a deterministic algorithm that blocks after spinning for time t. The randomized algorithm A from Theorem 10 blocks sometime before time t with probability $\pi(t)$. This function is differentiable at all t. Let p_t be the derivative of $\pi(t)$ at t. The expected cost of A on a lock wait σ equals $\int p_t C_{A_t}(\sigma)\, dt$. Because this cost is within a factor $e/(e-1)$ of *opt*'s cost, we have

$$\int p_t C_{A_t}(\sigma)\, dt \le \frac{e}{e-1} C_{opt}(\sigma).$$

Averaging over lock waits $\sigma(\mathscr{P})$ generated according to $\mathscr{P}$ yields

$$\int p_t \mathbf{E}C_{A_t}(\sigma(\mathscr{P}))\, dt \leq \frac{e}{e-1} \mathbf{E}C_{opt}(\sigma(\mathscr{P})).$$

Now there must be some A_t such that

$$\mathbf{E}C_{A_t}(\sigma(\mathscr{P})) \leq \frac{e}{e-1} \mathbf{E}C_{opt}(\sigma(\mathscr{P})),$$

because otherwise the integral would exceed the required bound. Let C^* be the infimum over all t of $\mathbf{E}C_{A_t}(\sigma(\mathscr{P}))$. Clearly, this value does not exceed $e/(e-1)$ times *opt*'s expected cost.

Let $\hat{A}$ be an adaptive algorithm that collects statistics on the lock-waiting times. Based on these statistics, it can choose an algorithm A_t that minimizes the expected cost on requests generated according to the observed distribution. (This algorithm spins for a time that equals either zero or some lock-wait time that has been observed.) It can then use this algorithm in handling the next lock wait. By the law of large numbers, the observed distribution will converge to the true distribution. This means that $\hat{A}$'s expected cost on a lock wait will approach C^*. Now suppose $\hat{A}$'s expected cost on a lock wait falls below $(1+\varepsilon)C^*$. Algorithm $\hat{A}$ is then competitive within ratio $(1+\varepsilon)e/(e-1)$ on all subsequent lock waits. An additive constant depending on ε can cover $\hat{A}$'s expected cost up to that point. It follows that $\hat{A}$ is c-competitive for all $c > e/(e-1)$. □

The convergence of this adaptive algorithm to $e/(e-1)$-competitive behavior depends on the fact that accurate statistics can be generated by keeping track of the entire history of lock-waiting times. A practical alternative to this algorithm, similar to that for snoopy caching, is one which only uses the last (or perhaps last few) lock-waiting times in order to determine what to do the next time a process requests a lock.

A simple adaptive algorithm A can decide how long a process should spin by using the length of time τ that the lock last remained held. If $\tau < C$, then the process spins for a time equal to C, otherwise the process blocks immediately. Note once again that this is an instance of choosing the algorithm that minimizes the expected cost, under the assumption that the lock-waiting time is equal to τ with probability 1. Furthermore, algorithm A is 3-competitive against all adversaries. The proof of this fact is virtually identical to the proof of Theorem 8.

In practice, the most commonly implemented strategy is to block immediately if the lock is not available, always incurring cost C. It is desirable that an adaptive algorithm not cost substantially more than this simple strategy. Assuming a certain independence in the way processors acquire locks, we can show that competitive advantage can be traded off against a guarantee that the adaptive algorithm does not perform too much worse than the algorithm that always blocks.

We define an algorithm A_α, based on a constant α, with $0 \leq \alpha \leq 1$, that determines the competitiveness and the bound on waiting time. As before, the time that a process spins before blocking depends on τ, the time the lock was last held. If $\tau \leq \alpha C$, the process spins for a time equal to αC and then blocks. Otherwise the process blocks immediately. A large value of α gives a more competitive algorithm, while a smaller value guarantees that the average waiting time is not much worse than C.

Now assume that the distribution of waiting times is nonincreasing, that is, the probability of waiting between a and b time units is at least as large as the probability of waiting between $a + k$ and $b + k$ time units. This would be the case if the times of lock requests in one process are independent of the times when locks are held by other processes, which is likely to be nearly true in many applications.

Let $f(t)$ be the probability that the waiting time is at most t. With probability $p = 1 - f(\alpha C)$, algorithm A_α blocks immediately. With probability $1 - p = f(\alpha C)$ it spins until either the lock is released or until time αC, whichever comes first. Clearly, $f(0) = 0$ and $f(\alpha C) = 1 - p$. If the distribution of waiting times is nonincreasing, $f(t)$ must be a least $t(1 - p)/\alpha C$ for all t from 0 to αC.

The expected cost of algorithm A_α is

$$\mathbf{E}C_{A_\alpha} = pC + (1 - p)\left(\int_0^{\alpha C} (1 - f(t)\, dt) + pC\right).$$

The first term is the expected cost covering the case where the previous lock was held for time longer than αC. The second term covers the case where the previous lock was released at or before time αC. It includes the expected cost of spinning at times 0 through αC as well as the cost of blocking if the current lock is held for time longer than αC.

Using the lower bound on $f(t)$, it follows that

$$\mathbf{E}C_{A_\alpha} \leq pC + (1 - p)\left((1 + p)\frac{\alpha C}{2} + pC\right).$$

The value of p for which this cost is maximized can be determined by differentiation to be

$$p = \frac{2}{2 + \alpha}.$$

Substituting in the previous formula and simplifying gives

$$\mathbf{E}C_{A_\alpha} \leq C\left(1 + \frac{\alpha^2}{2\alpha + 4}\right).$$

By choosing $\alpha = 0.5$, for example, the expected cost of the adaptive algorithm is bounded by $1.05C$.

5. The Two-Server problem. The k-server problem concerns planning the motion of k mobile servers in a metric space M. A request sequence σ specifies points in the metric space that require service. When a given point is requested, an algorithm must decide how to move the servers such that the given point is covered by some server. The cost of the request equals the sum of the distances moved by the servers. If the point was already covered at the time of the request, the algorithm may decide to move no servers at all, incurring zero cost. The k-server problem is a generalization of several important scheduling problems, including paging, caching, and disk-head motion planning [16].

Deterministic k-server algorithms have been examined in a number of recent studies. Manasse *et al.* [16] introduced the k-server problem and presented k-competitive algorithms for the cases when $k = 2$ and when $k = n - 1$, where n is the number of points in the metric space. They showed that k is a lower bound on the competitive factor of any deterministic algorithm, which implies that these algorithms are optimally competitive. Chrobak and Larmore [6] later presented a different approach to the two-server problem that gave another optimally competitive algorithm. Algorithms with competitive factor k have also been obtained for k servers when the metric space is uniform (i.e., there is unit distance between all points) [20], a line [5], or a tree [7]. It remains an open problem whether there is a k-competitive algorithm for all k on general metric spaces. Fiat *et al.* [11] showed that competitive algorithms exist for all k, although only within a factor much larger than k.

Randomized algorithms have also been studied. One collection of results concerns randomized algorithms against an adaptive on-line adversary. Raghavan and Snir [19] showed that k is a lower bound on the competitive ratio for these algorithms. They also presented a memoryless randomized algorithm (the harmonic algorithm) that achieves a ratio of 6 for the two-server problem. Berman *et al.* [2] showed that the same algorithm is competitive for three servers, within a very large factor, and Grove [12] later showed that it is $O(k2^k)$-competitive for k servers. Coppersmith *et al.* [8] obtained a k-competitive k-server algorithm whenever the metric space is finite and has a "resistive inverse."

The results we present in this section concern randomized algorithms against an oblivious adversary. The previous work in this area has focused on the paging problem, which is equivalent to the k-server problem on a uniform metric space. Fiat *et al.* [10] showed that there is a lower bound of H_k for this problem, where $H_k = \sum_{i=1}^{k} 1/i$ is the kth harmonic number, which is about $\ln(k)$. They presented algorithms that achieve this bound when $k = 2$ or when $k = n - 1$, where n is the number of points in the metric space. They also described an algorithm that is $2H_k$-competitive for all k. McGeoch and Sleator [18] later obtained an H_k-competitive algorithm for all k.

A natural question that arises is whether H_k-competitive algorithms exist for the k-server problem when the metric space is not uniform. The theorems in this section show that the answer to this question is no, by establishing upper and lower bounds on the competitive ratio for the two-server problem on certain triangles.

The techniques used in this work were later extended by Karloff *et al.* [15], who proved an $\Omega(\log \log k)$ lower bound on the competitive factor for the k-server problem on metric spaces with at least $k + 1$ points and an $\Omega(\log k)$ lower bound on the competitive factor for the k-server problem on sufficiently large metric spaces.

THEOREM 12. *Consider the two-server problem on an isosceles triangle with integer edge-lengths* 1, d, *and* d. *No algorithm is competitive against an oblivious adversary within a factor less than*

$$\frac{e_{2d-1} + 1/4d}{(e_{2d-1} - 1) + 1/2d}.$$

There is a randomized algorithm that achieves this competitive factor.

PROOF. We show that no phase-based on-line algorithm for this problem has an LP bound less than the constant given in the theorem. Together with Theorem 3, this implies the lower bound on the competitive factor of any algorithm. We also present a randomized algorithm that achieves this factor.

Suppose the points are a, b, and c, where the distance between a and b is 1, and where c is distance d away from the other points. We define our phases so that a phase ends whenever the locations of *opt*'s servers are known and one of those servers is at c. With the possible exception of the first phase, every phase begins and ends with *opt* known to be in state $\{a, c\}$ or known to be in state $\{b, c\}$. If the initial state is $\{a, b\}$, there is a special initial phase. We ignore this initial phase in the lower-bound proof, and we later deal with it easily in the upper-bound proof.

Let σ_k denote a phase of requests consisting of k requests alternating between a and b, followed by a request at c and then possibly one more request at a or b. We can easily compute the cost of an optimal algorithm *opt*. If $k < 2d$, *opt* will shuttle a server back and forth between a and b throughout the phase, with a total cost of k. When the request at c arrives, *opt*'s state is known and the phase ends. If $k \geq 2d$, *opt* will move on the first request from c to whichever of a or b is not already covered. When the request at c arrives, *opt* covers c from whichever of a or b is not the next request. After the request at a or b, *opt*'s state is known and the phase ends, with *opt* having total cost $2d$.

If an on-line algorithm A is phase-based, it must make its decisions on the basis of the number of requests to a and b so far in the phase. For each k from 1 to $2d$, let π_k denote the probability that A is covering both a and b after k requests. After $2d$ requests, *opt* is known to be in state $\{a, b\}$, so A must be in the same state. We can assume that A is lazy within any phase, making no moves if a requested point is covered and moving one server in response to a request at an uncovered point. (Laziness can only help an algorithm. For an explanation, see [16].) Laziness implies that $0 \leq \pi_1 \leq \cdots \leq \pi_{2d-1} \leq \pi_{2d} = 1$.

The expected cost of algorithm A on phase σ_k, where $k < 2d$, equals

$$(\pi_k)2d + \sum_{i=1}^{k} (1 - \pi_i).$$

The first term is the expected cost of moving back and forth along the long edges of the triangle. The ith term of the summation is the probability that a move across the short edge occurs on the ith request. Alternatively, if $k \geq 2d$, A's expected cost on the k requests at a and b is

$$d + \sum_{i=1}^{2d-1} (1 - \pi_i).$$

To minimize its expected cost on the final a or b request, A must cover c by moving from a and b with equal probability. This gives A expected cost $d + \frac{1}{2}$ on the last two requests of the phase.

If A has an LP bound of α, it must be that

$$\begin{cases} \mathbf{E}C_A(\sigma_k) = (\pi_k)2d + \sum_{i=1}^{k} (1 - \pi_i) \leq \alpha \cdot k, & \text{if } k < 2d, \\ \mathbf{E}C_A(\sigma_k) = 2d + \sum_{i=1}^{2d-1} (1 - \pi_i) + \frac{1}{2} \leq \alpha \cdot 2d & \text{otherwise.} \end{cases}$$

To establish a lower bound on the LP bound of any phase-based algorithm, we permit the π_k to be free variables and we minimize α subject to these constraints.

Variable π_1 has a positive coefficient on the left-hand side of one of the inequalities and a negative coefficient in all of the others. The positive coefficient appears in $(\pi_1)2d + (1 - \pi_1) \leq \alpha$. Suppose that the minimum value of α was obtained where this inequality was not tight. We could raise π_1 until the inequality was tight, which would only decrease the left-hand side of the other bounds. Thus we can replace the inequality with an equation without affecting the minimum α. Similarly, each other π_k appears with a positive coefficient in one inequality and a zero or negative coefficient in the rest. So for each k from 1 to $2d - 1$ we can replace an inequality by an equation, yielding

$$(*) \qquad (\pi_k)2d + \sum_{i=1}^{k} (1 - \pi_i) = \alpha k.$$

Straightforward transformation of the equations shows that

$$\pi_k = \frac{\alpha - 1 + (\pi_{k-1})2d - 1}{2d}.$$

We can then verify that, for all k from 1 to $2d - 1$,

$$\pi_k = (\alpha - 1)\left(\left(\frac{2d}{2d-1}\right)^k - 1\right).$$

Now, only one inequality remains:

$$2d + \tfrac{1}{2} + \sum_{i=1}^{2d-1} (1 - \pi_i) \leq \alpha 2d.$$

We can arrange this inequality and apply (*) with $k = 2d - 1$ to get

$$\alpha \geq 1 + \frac{1}{4d} + \frac{\sum_{i=1}^{2d-1}(1 - \pi_i)}{2d} = 1 + \frac{1}{4d} + \frac{\alpha(2d-1) - (\pi_{2d-1})2d}{2d}.$$

We can now rearrange further to obtain

$$\alpha \geq \frac{(2d/(2d-1))^{2d-1} + 1/4d}{(2d/(2d-1))^{2d-1} - 1 + 1/2d} = \frac{e_{2d-1} + 1/4d}{e_{2d-1} - 1 + 1/2d}.$$

By setting this to an equality we obtain the minimum value of α. This is a lower bound on the LP bound of any phase-based algorithm. By Theorem 3, this is a lower bound on the competitive factor of any algorithm.

A phase-based algorithm that achieves this competitive factor can be obtained by simply using the π_k obtained in the minimization. It is easy to show that $0 \leq \pi_1 \leq \cdots \leq \pi_{2d} = 1$ and that a lazy algorithm can be constructed where the algorithm has probability π_k of covering a and b after k requests. This ensures that the algorithm's expected cost is within factor α of *opt*'s cost on each phase. (If the problem begins with a and b covered, the algorithm can ignore all requests until there are requests at c and then either a or b. Its cost for this partial phase is at most $d + 1$.) □

The competitive ratio obtained in Theorem 12 grows and approaches $e/(e - 1)$ as d grows.

The proof for isosceles triangles was greatly simplified by the symmetry of graph. The following theorem shows that the basic technique can be extended to other nonequilateral triangles, again yielding a optimal competitive factor that is larger than 3/2.

THEOREM 13. *Consider the two-server problem on a triangle with edge-lengths* 3, 4, *and* 5. *No algorithm is competitive against an oblivious adversary within a*

competitive factor less than $1652/1069 \approx 1.545$. *There is a randomized algorithm that achieves this competitive factor.*

PROOF. Let M be a metric space with three points, a, b, and c, where $d(ab) = 3$, $d(ac) = 5$, and $d(bc) = 4$.

Any request sequence can be partitioned into phases. The possible phases are listed below:

Initial state	Request sequence	Final state	*opt*'s cost
$\{a, b\}$	*ca*	$\{a, c\}$	4
$\{a, b\}$	*cbaba*	$\{a, b\}$	8
$\{a, b\}$	*cbabca*	$\{a, c\}$	12
$\{a, b\}$	*cbabcb*	$\{b, c\}$	11
$\{a, b\}$	*cbac*	$\{a, c\}$	8
$\{a, b\}$	*cbc*	$\{b, c\}$	5
$\{a, c\}$	*bab*	$\{a, b\}$	4
$\{a, c\}$	*bac*	$\{a, c\}$	6
$\{a, c\}$	*bc*	$\{b, c\}$	3
$\{b, c\}$	*aba*	$\{a, b\}$	5
$\{b, c\}$	*abc*	$\{b, c\}$	6
$\{b, c\}$	*ac*	$\{a, c\}$	3

Suppose A is a phase-based algorithm. A must use fixed probabilities in choosing where to place its servers, where the probabilities depend only on the requests in the current phase. At the end of a phase, A must be in the same configuration as *opt*. Nine probabilities suffice to describe A's behavior at other times. In the following table, the entry in the column for each state is the probability that A is in that state after starting in the given initial state and processing the given requests:

Initial state	Request sequence	Probability		
		$\{a, b\}$	$\{a, c\}$	$\{b, c\}$
$\{a, b\}$	*c*	0	π_1	$1 - \pi_1$
$\{a, b\}$	*cb*	π_2	0	$1 - \pi_2$
$\{a, b\}$	*cba*	π_3	$1 - \pi_3$	0
$\{a, b\}$	*cbab*	π_4	0	$1 - \pi_4$
$\{a, b\}$	*cbabc*	0	π_5	$1 - \pi_5$
$\{a, c\}$	*b*	π_6	0	$1 - \pi_6$
$\{a, c\}$	*ba*	π_7	$1 - \pi_7$	0
$\{b, c\}$	*a*	π_8	$1 - \pi_8$	0
$\{b, c\}$	*ab*	π_9	0	$1 - \pi_9$

We can assume that A is lazy within any phase. With this assumption, we can compute A's expected cost on each complete phase:

Initial state	Request sequence	Final state	A's cost
$\{a, b\}$	ca	$\{a, c\}$	$8 - 4\pi_1$
$\{a, b\}$	$cbaba$	$\{a, b\}$	$16 + 2\pi_1 - 4\pi_2 - 2\pi_3 - 4\pi_4$
$\{a, b\}$	$cbabca$	$\{a, c\}$	$14 + 2\pi_1 - 4\pi_2 - 2\pi_3 + 6\pi_4 - 4\pi_5$
$\{a, b\}$	$cbabcb$	$\{b, c\}$	$11 + 2\pi_1 - 4\pi_2 - 2\pi_3 + 6\pi_4 + 2\pi_5$
$\{a, b\}$	$cbac$	$\{a, c\}$	$8 + 2\pi_1 - 4\pi_2 + 6\pi_3$
$\{a, b\}$	cbc	$\{b, c\}$	$5 + 2\pi_1 + 6\pi_2$
$\{a, c\}$	bab	$\{a, b\}$	$10 - 4\pi_6 - 2\pi_7$
$\{a, c\}$	bac	$\{a, c\}$	$6 - 4\pi_6 + 6\pi_7$
$\{a, c\}$	bc	$\{b, c\}$	$3 + 6\pi_6$
$\{b, c\}$	aba	$\{a, b\}$	$11 - 2\pi_8 - 4\pi_9$
$\{b, c\}$	abc	$\{b, c\}$	$6 - 2\pi_8 + 6\pi_9$
$\{b, c\}$	ac	$\{a, c\}$	$3 + 6\pi_8$

If A has an LP bound of α, then the set of possible phases implies a set of constraints that must hold. For example, the first phase listed above implies that

$$8 - 4\pi_1 \leq 4\alpha + \Phi_{ab} - \Phi_{ac}.$$

The left-hand side of the inequality is A's expected cost, and the constant 4 on the right-hand side is *opt*'s cost. There are 12 possible phases and therefore 12 possible constraints.

To establish a lower bound on the LP bound of phase-based algorithms, we can permit the probabilities and potentials to be free variables and then minimize α subject to the 12 constraints described above. Minimizing α subject to these constraints, we obtain $\alpha = 1652/1069$. By Theorem 3, this is a lower bound on the competitive factor of any algorithm.

Now consider an algorithm A based the values determined in the minimization: $\pi_1 = 2161/3207$, $\pi_2 = 169/1069$, $\pi_3 = 1481/3207$, $\pi_4 = 2749/3207$, $\pi_5 = 1016/3207$, $\pi_6 = 371/1069$, $\pi_7 = 2491/3207$, $\pi_8 = 212/1069$, $\pi_9 = 1961/3207$. It is straightforward to verify that A can be constructed so that it is lazy. This means that its expected cost in each phase equals the cost used in the lower-bound proof, and therefore α is indeed A's LP bound. By Theorem 2, A must be α-competitive. □

There is another triangle that has the same competitive factor as the equilateral triangle, as shown by the following theorem.

THEOREM 14. *Consider the two-server problem on a triangle with integer edge-lengths* 1, 1, *and* 2. (*The points are collinear.*) *No algorithm is competitive within a factor less than* 3/2. *There is a randomized algorithm that achieves this competitive factor.*

PROOF. This can be proven rigorously in a proof similar to the previous one. Informally, there is a single situation in which the location of *opt*'s servers is unknown: If *opt* is covering the outer points and a request arrives at the middle one, *opt*'s response will be based on which point is requested next. The best response for a randomized algorithm is to move from the outer points with equal probability. Straightforward calculation of the algorithms' costs shows that the randomized algorithm will always have expected cost 3 in a phase in which *opt*'s cost is 2. □

For two-server problems on triangles, the smallest competitive ratios we have been able to obtain arise on the very simplest graphs: the equilateral triangle and the 1–1–2 triangle. In each case there is a single moment of decision for a randomized algorithm which is best handled by covering the possibilities equally. For the other triangles we have considered, there are more decision times and the competitive ratio increases. It remains an open problem to develop a complete understanding of randomized competitive ratios for the two-server problem. The triangles we have studied all have competitive ratios below $e/(e-1)$. Is this the limit for triangles? What is the limit for the two-server problem on general metric spaces? Karloff *et al.* [15] proved lower bounds for randomized k-server algorithms. Can matching upper bounds be obtained?

Acknowledgments. The authors thank Howard Karloff and Dan Velleman for many useful discussions and for ideas that improved the paper. We also gratefully acknowledge the contributions of an anonymous referee.

Appendix. A more general form of Theorem 1 can be obtained for server problems. In the same way that it is sometimes possible to "know" the optimal algorithm's state, it is sometimes possible to "know" the positions of some of *opt*'s servers. We can assume that the off-line algorithm is covering a set of points V_{opt} if the cost of being in each state s is at least the cost of being in some state s_0 where V_{opt} is covered plus the cost of a move from s_0 to s. We say that *opt is known to be covering* V_{opt} whenever this situation arises.

THEOREM 15. *If A is an α-competitive server algorithm, there must be an α-competitive algorithm A' that covers any set of points that opt is known to be covering.*

PROOF. Suppose that *opt* is known to be covering a set of points V_{opt} at time t. A cruel adversary can repeatedly request those points without increasing *opt*'s cost. These requests will either force A to cover V_{opt} or to incur large costs.

We can assume that A is lazy. Let V_t be the set of points that are initially covered or that are requested up to time t. Laziness implies that at time t, A covers a subset of the points in V_t. Let ε be the shortest distance between distinct points in V_t and let E be the longest.

Algorithm A' handles each request sequence σ by simulating A on a longer

sequence σ' which includes extra requests for points that are covered by *opt*. These extra requests do not increase *opt*'s cost. A' chooses its moves based on the moves made by A. No matter what choices A makes in handling σ', A' will be able to handle σ without extra cost. Summing over the probability distribution of A's possible actions, we get $\mathbf{E}C_{A'}(\sigma) \leq \mathbf{E}C_A(\sigma')$. Therefore, if A is α-competitive there is some a such that

$$\mathbf{E}C_{A'}(\sigma) \leq \mathbf{E}C_A(\sigma') \leq \alpha \cdot C_{opt}(\sigma') + a = \alpha \cdot C_{opt}(\sigma) + a,$$

and thus A' is α-competitive.

It remains to describe algorithm A'. In general A' passes each of its requests to the simulator of A and then mimics A's response. Whenever *opt* is known to be covering some set of points V_{opt}, $|V_{opt}| > 1$, algorithm A' gives some extra requests to the simulator. It cycles $2kE/\varepsilon$ times through requests for the entire set V_{opt}. After these requests, A may be covering all of V_{opt}. In this case A' jumps directly to A's final configuration. If A does not eventually cover all of V_{opt}, A' moves in any way to cover set V_{opt}. Immediately before handling the next request, A' moves to A's configuration.

The cost of algorithm A' is at most the cost that A incurs on the longer sequence. This is straightforward except when A ultimately fails to cover V_{opt}. In this case A's cost is high: at least ε on each pass through V_{opt}. There are $2kE/\varepsilon$ passes, yielding total cost $2kE$. This is enough to cover the cost of A''s moves to V_{opt} (at most kE) and then to A's position (at most kE).

(If the metric space is finite, ε and E can be defined to be the shortest and longest distances between distinct points of the metric space. This definition is independent of the choice of request sequence and bounds the length of the simulation.) □

References

[1] Ben-David, S., Borodin, A., Karp, R., Tardos, G., and Wigderson, A. On the power of randomization in on-line algorithms. *Algorithmica*, **11**:2–14, 1994.

[2] Berman, P., Karloff, H., and Tardos, G. A competitive three-server algorithm. *Proceedings of the First Annual ACM–SIAM Symposium on Discrete Algorithms*, San Francisco, 1990, pages 280–290.

[3] Borodin, A., Linial, N., and Saks, M. An optimal online algorithm for metrical task systems. *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, New York, 1987, pages 373–382.

[4] Borodin, A., Linial, N., and Saks, M. An optimal on-line algorithm for metrical task systems. *J. Assoc. Comput. Mach.*, **39**(4):745–763, 1992.

[5] Chrobak, M., Karloff, H., Payne, T., and Vishwanathan, S. New results on server problems. *SIAM J. Discrete Math.*, **4**(2):172–181, 1991.

[6] Chrobak, M. and Larmore, L. L. A new approach to the server problem. *SIAM J. Discrete Math.*, **4**(3):323–328, 1991.

[7] Chrobak, M. and Larmore, L. L. An optimal on-line algorithm for k servers on trees. *SIAM J. Comput.*, **20**(1):144–148, 1991.

[8] Coppersmith, D., Doyle, P., Raghavan, P., and Snir, M. Random Walks on Weighted Graphs, and Applications to On-Line Algorithms. *J. Assoc. Comput. Mach.*, **40**(3):421–453, 1993.

[9] Eggers, S. J., and Katz, R. H. Evaluating the performance of four snooping cache coherency

protocols. *Proceedings of the 16th Annual International Symposium on Computer Architecture*, 1989.

[10] Fiat, A., Karp, R. M., Luby, M., McGeoch, L. A., Sleator, D. D., and Young, N. E. Competitive paging algorithms. *J. Algorithms*, **12**(4):685–699, 1991.

[11] Fiat, A., Rabani, Y., and Ravid, Y. Competitive k-server algorithms. *J. Comput. System Sci.*, to appear.

[12] Grove, E. F. The harmonic online k-server algorithm is competitive. *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, New Orleans, 1991, pages 260–266.

[13] Irani, S., and Rubinfeld, R. A competitive 2-server algorithm. *Inform. Process. Lett.*, **39**(2):85–91, 1991.

[14] Karlin, A. R., Manasse, M. S., Rudolph, L., and Sleator, D. D. Competitive snoopy caching. *Algorithmica*, **3**(1):79–119, 1988.

[15] Karloff, H., Rabani, Y., and Ravid, Y. Lower bounds for randomized k-server and motion-planning algorithms. *SIAM J. Comput.*, to appear, 1994.

[16] Manasse, M. S., McGeoch, L. A., and Sleator, D. D. Competitive algorithms for server problems. *J. Algorithms*, **11**(2):208–230, 1990.

[17] McGeoch, L. A. Algorithms for Two Graph Problems. Ph.D. thesis, Carnegie Mellon University, 1987.

[18] McGeoch, L. A., and Sleator, D. D. A strongly competitive randomized paging algorithm. *Algorithmica*, **6**(6):816–825, 1991.

[19] Raghavan, P., and Snir, M. Memory Versus Randomization in On-line Algorithms. IBM Research Report, 1990. Submitted for publication.

[20] Sleator, D. D., and Tarjan, R. E. Amortized efficiency of list update and paging rules. *Comm.*