# Multiple Spin-Block Decisions*

Peter Damaschke
School of Computer Science and Engineering
Chalmers University
41296 Göteborg, Sweden
ptr@cs.chalmers.se

**Abstract**

We study the online problem of holding a number of idle threads on an application server, which we have ready for processing new requests. The problem stems from the fact that both creating/deleting and holding threads is costly, but future requests and completion times are unpredictable. We propose a practical scheme of barely random discrete algorithms with competitive ratio arbitrarily close to $e/(e-1)$, where $e \approx 2.718$ is Euler's number. The competitive ratio is sharply concentrated for any input. The results are generalizations of the well-known result for the rent-to-buy problem.

**Key words:** multithreading, spin-block problem, online algorithms, randomization, implementation issues

*An extended abstract appeared in the Proceedings of the *10th International Symposium on Algorithms and Computation ISAAC'99*, Chennai/India, *Lecture Notes in Computer Science* 1741 (Springer), pp. 27-36.

# 1 Introduction

**Problem statement.** A server in a network has to execute a large number of jobs due to requests from several clients. Multithreading is an approach to serve many requests simultaneously, either on parallel processors or scheduled by a multitasking operating system, such that short requests do not have to wait for completion of other time consuming jobs, which would be frustrating or even unacceptable. Each arriving job is assigned to some currently idle thread which is responsible for completing the job. If no thread is idle then a new thread is created. (We consider systems not refusing jobs.) A busy thread that has finished a job becomes idle again.

The workload of the server, i.e. the number of simultaneous jobs, can fluctuate over time very much, but creating and deleting a thread is a costly operation. In object-oriented environments, this is more expensive than creating/deleting other kinds of object, cf. [8]. So we should have enough idle threads ready for future requests. On the other hand, running many threads on the spot over long periods would be a waste of processor cycles which could be better used by other applications residing on the same machine. (The scheduler periodically checks all threads, also idle ones, whether some work is to be done.) Hence one has to observe a suitable strategy for deleting idle threads, so as to keep the total costs possibly small, without knowledge of future jobs. This is a typical online problem which can be stated as follows: How should we assign new jobs to idle threads and which of the idle threads should we delete at what time, in order to minimize the total costs for creating, deleting, and running threads? (We presume some familiarity with online algorithms, the concept of competitiveness, and the several notions of adversaries; see [1] for a general introduction to the field.)

The problem is sometimes implicitly mentioned in articles in software developer magazines, as the idea of pooling threads and deleting them after some idle time is quite common. For heavily loaded servers with fluctuating density of calls, savings on this front may have a measurable effect [2]. The subject is particularly interesting if the request sequence mainly consists of clusters of many short jobs, which is a typical case in many environments. In view of this practical motivation, the strategy should not only be competitive with respect to the mentioned costs, but also easy to implement and, most importantly, it must be computationally simple, to not compensate savings of thread costs by large amount of additional data structure for thread administration.

Although our motivation was thread administration on a server, other fields of interest are imaginable, too. To mention an example, the same problem appears when

a pool of database connections is left open for serving future requests. (Opening and closing a connection is expensive). One may further think of supplying energy if switching the sources on and off incurs some costs, or of renting storage space in a warehouse if a fee is charged for each alteration of the contract. See [7] for more hints.

**Model.** We assume that progress of busy threads within our server application does not depend on the number of additional idle threads, but they take available processor time that might else be used by other applications. (This is suitable e.g. if the system devotes some fixed fraction of time slices to the busy threads, or if threads are delegated to different processors on a parallel machine.) In other words, we assume that the schedule of jobs is given to the online player and is beyond his influence. In particular, the workload, i.e. the number of simultaneous jobs which equals the number of threads that have to be busy, is given at any time.

We assume the following costs. Running an idle thread one time unit long incurs cost 1, reflecting a penalty for the scheduler's extra work. Since time and cost units can be chosen arbitrarily, we may w.l.o.g. fix them in such a way that $C + D = 1$, where $C$ and $D$ are constant costs of each create and delete operation, respectively. That means, running a thread for 1 time unit is as expensive as a creation-deletion pair. We will find this choice to be very convenient. (The time unit is fixed henceforth. Prior to real installation on a concrete machine, the cost ratio, and hence the appropriate time unit, must be estimated or determined by experiment which may be part of the installation routine.)

**Overview.** As we shall see in Section 2, our problem turns out to be a certain natural generalization of the single rent-to-buy problem, which is, along with some of its variants, well-known under different names, including the leasing problem, spin-block problem, ski rental problem etc.: One needs a resource for a time interval with previously unknown duration. One is allowed to rent the resource at price 1 per time unit or to buy it at an arbitrary moment at price 1. To distinguish our problem from single rent-to-buy, we refer to it as the multiple spin-block problem. (Let an idle thread spin, or block/delete it.)

Quite trivially, the best deterministic competitive ratio for single rent-to-buy is 2. In Section 2 we give an optimal offline algorithm, called BRIDGES, for our problem, and in Section 3 we propose two different 2-competitive deterministic online algorithms.

In contrast to the optimal deterministic result, there exists an $e/(e-1)$-competitive randomized algorithm against an oblivious adversary [6], where $e \approx 2.7183$ is Euler's

number. Throughout the paper it is called the KMMO algorithm. We shall briefly review the KMMO algorithm in Section 4, and then we describe how any randomized rent-to-buy algorithm can be turned into one for the mulitple spin-block problem, preserving the expected competitive ratio. However the drawback of the resulting algorithm is that it either needs much randomness, or the actual competitive ratio can be much higher than its expectation.

In order to get algorithms with smoother behaviour, we first study discrete randomized rent-to-buy algorithms in Section 5. Such an algorithm may buy the resource only in a finite set of points on the time axis (unlike e.g. the KMMO algorithm that can buy at any time according to a certain probability density). The main result of Section 5 is that some $n$-point discretization of the KMMO algorithm has competitive ratio no worse than $e/(e-1)+1/2n$, and this is optimal, subject to an $1/n^2$ term. In Section 6 we use this result to construct a scheme of $r$-competitive algorithms for multiple spin-block decisions, with $r$ arbitrarily close to $e/(e-1)$. It is computationally simple, guarantees a sharply concentrated competitive ratio, and it is barely random in the sense that a constant number (depending on $n$) of random bits per time unit is used, for arbitrarily many simultaneous threads.[1]

In Section 7 we briefly mention some extensions of our results: algorithms with lookahead, and the influence of inaccurate cost measures on the competitiveness.

**Related literature.** As mentioned, our results are based on the optimal randomized online algorithm for rent-to-buy [6]. A time-discrete version has been given also in [3] in a more general context (leasing with interest rates), however for our purpose we need solutions with prescribed probabilities $1/n$. In [7] a sequence of isolated rent-to-buy decisions is studied under the assumption that requests follow an unknown but fixed probability distribution, and the goal is to adapt the online player's strategy to this distribution. In contrast, we consider concurrent threads which overlap in time, and we do not make probabilistic assumptions. The TCP acknowledgment delay problem [4] is of similar flavour as ours. An essential difference is that each acknowledgment has unit cost regardless the number of acknowledged packets, whereas our operations create or delete only one thread each. The call admission problem is also very different from ours, as requests may be rejected due to limited capacities, and the goal is to maximize the throughput (see e.g. [5]).

[1] Usually, the term "barely random" refers to online algorithms, processing a sequential input, that flip constantly many random coins in the beginning and are otherwise deterministic. But since our input has unlimited size also at any time, it seems appropriate to adopt this phrase.

## 2 Decomposition of the Multiple Spin-Block Problem

First we observe that only the number of running jobs and threads, respectively, at any time is relevant to the costs: Since idle threads are identical, it is not essential which of them is selected to serve a new job or deleted. (This may only affect the administration costs, however this matter can be considered separately.) Similarly, the start and end times of individual jobs are not relevant, we only need to consider their total number as a function of time.

This staircase function from the real numbers (time) into the nonnegative integers (number of running jobs) is called the *workload function* $f$. It is not a restriction to assume that no jobs start or end at exactly the same moment, so $f$ always increases/decreases by at most 1. For competitive analysis we consider a finite amount of work, thus we have $f = 0$ outside the finite interval $I$ from the arrival of the first job until completion of the last job. Function $f$ is the input of our online problem, and the online player always knows $f$ up to the current moment, but not the future part. The outcome of the desired algorithm is nothing else than a staircase function $g \geq f$ indicating the number of (busy and idle) threads at each time. (By the above discussion, the assignment of jobs to threads can be considered as an independent matter.)

Next we introduce some useful notion. We say that a staircase function $h$ has a *downwards step* at time $t$ if $h$ decreases at $t$. We say that $h$ has an *upwards step* at time $t$ if $h$ increases there. The *ordinate* of a downwards step is defined to be the function value after that step. The ordinate of an upwards step is defined to be the function value before that step. A *down-up pair* of $h$ is a downwards step together with the next upwards step of $h$ which has the same ordinate. (See Fig. 1.) The ordinate of a down-up pair is the ordinate of its downwards and upwards step. Note that the down-up pairs can be considered as matching open and close parentheses, like in an arithmetic expression. The *width* of a down-up pair $P$, denoted $w(P)$, is the distance between its downwards step at time $d(P)$ and upwards step at time $u(P)$, that is $w(P) = u(P) - d(P)$. Note that exactly $\max_t h(t)$ upwards and downwards steps, respectively, are not involved in down-up pairs. Nevertheless, it is convenient to consider them as $\max_t h(t)$ down-up pairs of infinite width. (This will avoid a few tiresome case distinctions in subsequent discussions.)

Remember that we are interested in the costs of create/delete operations and running idle threads. For a given result function $g$, these costs obviously consists of the following summands:

- $C$ times the number of upwards steps of $g$,
- $D$ times the number of downwards steps of $g$,
- the area between the graphs (staircase curves) of $g$ and $f$.

Due to our convention $C + D = 1$, the first two terms can be replaced with the number of down-up pairs of $g$ (including those of infinite width.)

The optimum (offline) cost for a workload function $f$ is the minimum cost of $g$ such that $g \geq f$. (Note that, as with $f$, we have $g = 0$ outside some finite interval.) While an offline algorithm may fix $g$ with full knowledge of $f$, an online algorithm can use values of $f$ up to $t$ only, to fix $g(t)$.

Consider any staircase function $g \geq f$. If $g$ has some upwards step at $t$ such that $g > f$ immediately after $t$, we can postpone this upwards step, without adding new down-up pairs to $g$, thereby reducing the area below the graph of $g$. We conclude:

**Observation 2.1** *An optimum $g$ has upwards steps only at such $t$ where $f$ has upwards steps, too, and these corresponding upwards steps of $f$ and $g$ always have the same ordinate (hence $f = g$ around $t$). An equivalent condition is: Whenever we get $g > f$ at some moment, $g$ is then monotone decreasing at least until $g = f$ is reached next.*

Now let $g$ be a function satisfying the condition mentioned above. For every down-up pair $P$ of $f$, define $s(P)$ such that $d(P) + s(P)$ is the time $g$ reaches the ordinate of $P$ again before $u(P)$ (maybe $s(P) = 0$), and $s(P) = u(P) - d(P)$ otherwise, i.e. if $g > f$ during the whole interval from $d(P)$ to $u(P)$. (Fig. 1 contains all these cases.) From Observation 2.1 it is not hard to see that $g(t) = f(t) + \#\{P : \ d(P) \leq t < d(P) + s(P)\}$ for all times $t$.

It follows that we can restrict attention to functions $g$ constructed in the following way:

**Observation 2.2** *An optimum $g$ satisfies*

$$g(t) = f(t) + \#\{P : \ d(P) \leq t < d(P) + s(P)\},$$

*where the $s(P) \leq w(P)$ are nonnegative numbers assigned to the down-up pairs $P$ of $f$.*

This is used to decompose the multiple spin-block problem: We charge the down-up pairs of $f$ with costs which sum up to the costs of $g$:

**Definition 2.3** *The cost of a down-up pair $P$ of $f$ is $s(P)$ if $s(P) = w(P)$, and $1+s(P)$ if $s(P) < w(P)$.*

This "pricing" is justified by

**Lemma 2.4** *For functions $g$ as constructed above, the cost of $g$ is the sum of costs contributed by all down-up pairs of $f$.*

**Proof.** The area between the graphs of $g$ and $f$ is obviously $\sum_P s(P)$. By construction, the only downwards steps of $g$ are at times $d(P) + s(P)$, for such $P$ with $s(P) < w(P)$, whereas in case $s(P) = w(P)$, pair $P$ does not affect the course of $g$. Thus we add cost 1 (for a down-up pair of $g$) for each $P$ with $s(P) < w(P)$. □

We specify an offline algorithm called BRIDGES by:

**Definition 2.5** *BRIDGES is given by $s(P) = w(P)$ if $w(P) \leq 1$, and $s(P) = 0$ if $w(P) > 1$.*

We have chosen this name since, figuratively speaking, the graph of $g$ builds bridges over all valleys of $f$ not longer than 1, and $g = f$ elsewhere. Note that the lookahead of BRIDGES is bounded by one time unit.

**Lemma 2.6** *For every workload function, BRIDGES yields the unique optimal solution, with cost $\sum_P \min\{1, w(P)\}$.*

**Proof.** We must show that any $g$ is more expensive than that chosen by BRIDGES. As we have seen, we may w.l.o.g. assume that $g$ is constructed from numbers $s(P)$, and the total cost is given by Lemma 2.4. Hence it is enough to show that only the $s(P)$ as chosen by BRIDGES lead to the smallest possible cost of every down-up pair $P$. In fact, $P$ has cost $\min\{w(P), \min_{s(P)<w(P)}(1 + s(P))\} = \min\{w(P), 1\}$. Obviously this is minimized for $s(P) = w(P)$ if $w(P) \leq 1$, and $s(P) = 0$ if $w(P) > 1$. □

So BRIDGES is an optimal offline algorithm. Clearly, an online algorithm is not able to choose $s(P)$ in this way, since $w(P)$ is not known at time $d(P)$. However we achieve competitive ratio 2 in the next section.

## 3 Strongly Competitive Deterministic Algorithms

In the first theorem of this section we are going to describe an online algorithm in terms of the values $s(P)$. The question how to achieve these values by an implementation is discussed afterwards.

**Theorem 3.1** *The algorithm specified by* $s(P) = \min\{1, w(P)\}$ *is 2-competitive.*

**Proof.** First note that we have indeed an online algorithm: For fixing $g$ at any time it suffices to know whether the contribution of each $P$ is still 1 or already 0. Rule $s(P) = \min\{1, w(P)\}$ means that this summand remains 1 until $d(P) + 1$. There is no need to know $w(P)$ in advance.

For each $P$, the algorithm pays $w(P)$ if $w(P) \leq 1$, and 2 if $w(P) > 1$. This is at most twice the optimum cost $\min\{1, w(P)\}$. Thus Lemma 2.4 and 2.6 imply 2-competitiveness. □

The above algorithm is a natural generalization of the optimal (2-competitive) rent-to-buy strategy, which we apply to every down-up pair of $f$. To "rent" means to let an idle thread run, and to "buy" means to delete an idle thread (and create a new one later if needed). Note that a workload function with $f \in \{0, 1\}$ everywhere leads to nothing but a sequence of independent instances of the rent-to-buy problem.

For a concrete thread administration (our original motivation) we have to specify which threads are deleted resp. allocated to new jobs, rather than fixing their number $g$ only. It is fairly obvious that the following algorithm implements the strategy of Theorem 3.1. (Be careful to distinguish between "delete" and "remove".)

**EXPIRATION DATE STACK**

Maintain a stack of threads.
(1) When a thread becomes idle at time $t$, assign expiration date $t + 1$ to it, and add it to the stack.
(2) When the expiration date of the thread at the bottom of the stack is reached, delete this thread and remove it from the stack.
(3) When a new job arrives, assign it to the top idle thread on the stack and remove the now busy thread from the stack.

Note that the expiration dates are monotone on the stack, hence the thread to expire next is always the bottom element. (Although (3) is not a stack operation, we use the term "stack" to stress the fact that idle threads are added and assigned to jobs on a last-in-first-out basis.)

**Corollary 3.2** *EXPIRATION DATE STACK is* 2*-competitive.* □

EXPIRATION DATE STACK is easy enough to implement, however it must maintain a stack of threads with their expiration dates. We have another, simpler algorithm that needs to store only two numbers in order to compute the next expiration date. The number of threads is now denoted $h$.

## CUMULATIVE IDLE COSTS

Let initially $t_0$ be the left endpoint of $I$, and $t$ the current time.
Whenever $\int_{t_0}^{t}(h(x) - f(x))dx = 1$, delete one thread and reset $t_0 := t$. Reset $t_0$ to $t$ also whenever $h(t) = f(t)$.

Note that the integral is the cost of idle threads between $t_0$ and $t$. Clearly, instead of permanently updating the integral we can easily precompute the moment when it will achieve 1 if the number $i$ of idle threads does not further change, and this must be updated only when $i$ does change yet.

Although both algorithms behave quite different on the same instance, the worst-case performance is the same:

**Theorem 3.3** *CUMULATIVE IDLE COSTS is* 2*-competitive.*

**Proof.** Let $f$ be the workload function, $g$ the function that BRIDGES would produce, and $h$ the function produced by CUMULATIVE IDLE COSTS. Consider any down-up pair $P$ of $h$, and let $t_0$ be the moment of its downwards step. We charge $P$ with cost 2, namely $C + D = 1$ for the downwards and upwards step, and 1 for the idle threads until the next expiration date $t$ (i.e. the integral). So the intervals $[t_0, t]$ of different $P$ partition the time axis, such that the costs of $h$ are bounded by twice the number of down-up pairs. It remains to compare this to the costs of $g$, which are optimal by Lemma 2.6.

First we look at such $P$ where $h(t_0) \geq g(t_0)$. Then the downwards step of $P$ at $t_0$ can be considered as a delayed downwards step of some down-up pair $Q$ of $g$, namely

the most recent one at the same ordinate. Observe that different $P$ correspond to different $Q$ (since $h$ increases only if $h = f$), and that BRIDGES pays at least 1 for each $Q$.

For $P$ with $h(t_0) < g(t_0)$ we use a different argument. Function $h$ makes a downwards step $P$ only after some time interval $[t_0, t]$ where $h$ was constant, and the size of the area between the graphs of $h$ and $f$ on $[t_0, t]$ has run up to 1. This area also lies between the graphs of $g$ and $f$ (since $h < g$). Moreover, these areas are disjoint for different $P$. On the other hand, the total cost incurred by BRIDGES is at least the area between the graphs of $g$ and $f$.

Altogether, CUMULATIVE IDLE COSTS pays at most twice the optimum. □

We remark that these deterministic algorithms are strongly competitive, since 2 is the optimal competitive ratio already for the special case of the rent-to-buy problem. In contrast to online scheduling problems (cf. [9]), knowledge of execution times of current jobs is not helpful to improve the competitive ratio, since an adversary may send arbitrarily short requests and extend completed jobs immediately by new ones, so the online player cannot exploit such knowledge.

# 4 Randomization

A competitive ratio below 2 can be obtained by randomization.

A randomized rent-to-buy algorithm is nothing else than a probability distribution on the points $b$ in time at which we shall buy the resource. In the KMMO algorithm [6], the probability to buy before $b$ is a continuous function, namely $\int_0^b (e-1)^{-1} e^t dt$. Under the assumption that uniformly distributed random reals $X$ from $[0, 1]$ are available, we may set $b = \ln(1 + (e-1)X)$.

In the following, $R$ is an arbitrary but fixed randomized rent-to-buy algorithm with expected competitive ratio $r$ (against an oblivious adversary), buying the resource after expected time $r-1$. For example, take the KMMO algorithm with $r = e/(e-1) \approx 1.58$.

An obvious randomized version of CUMULATIVE IDLE COSTS might come first into mind:

**CUMULATIVE IDLE COSTS ($R$)**

Let initially $t_0$ be the left endpoint of $I$, and $t$ the current time. Fix $b$ according to $R$. Whenever $\int_{t_0}^t (h(x) - f(x))dx = b$, delete one thread, reset $t_0 := t$, and sample a new $b$.

Unfortunately this attempt fails, as the following heuristic consideration shows.

**Observation 4.1** *CUMULATIVE IDLE COSTS (R) is not even 2-competitive.*

To see this, we construct an example of a function $f$ that fools the algorithm. We discuss only the part of $f$ being essential for our claim. First $f$ grows to $n$, then $n$ downwards steps follow at infinitesimal distances, and $t < 1$ time units later $f$ makes $n$ upwards steps, also in quick succession. Figuratively speaking, $f$ has a "canyon" of depth $n$ and width $t$. Since $t < 1$, BRIDGES would pay $tn$ to cross the canyon. Now, let $h$ be the (random) function produced by CUMULATIVE IDLE COSTS ($R$). We assume the best case that $h = f$ at the left edge of the canyon. The idea of the example is that $h$ will typically have unnecessary costs due to many downwards steps inside the canyon (and their corresponding upwards steps). The area between $h$ and $f = 0$ accumulates to 1 after time $1/n$. Since $R$ has expected competitive ratio $r$, function $h$ steps down after expected time $(r-1)/n$. Iterating this argument, $k$ downwards steps are performed after expected time $\sum_{i=n-k+1}^{n}(r-1)/i$. The expected area below $h$ until this moment is $k(r-1)$. For large enough $n$ we may take the expectations as true values. Hence the number $k$ of downwards steps satisfies $\sum_{i=n-k+1}^{n}(r-1)/i \approx t$. Let be $y = t/(r-1)$. Since the harmonic numbers grow as ln, this gives $\ln n - ln(n-k) \approx y$, hence $k \approx n(1-e^{-y})$. For small $t$ this simplifies to $k \approx nt/(r-1)$. CUMULATIVE IDLE COSTS ($R$) pays $k$ for the steps and $k(r-1)$ for the area, a total of $kr$, compared to $tn$ payed by BRIDGES. For $f$ composed of a chain of such narrow canyons, these are the dominating costs, thus the expected competitive ratio becomes $kr/tn \approx r/(r-1)$. Ironically, this is larger than 2 just because of $r < 2$.

This example suggests to pay attention to the moments when the threads became idle, which means that randomizing EXPIRATION DATE STACK is the proper way.

**EXPIRATION DATE STACK ($R$), Version 1**

Maintain a stack of threads.
(1) When a thread becomes idle at time $t$, assign expiration date $t + b$ to it, where $b$ is chosen according to $R$, and add it to the stack.
(2) When the expiration date of a thread is reached, delete this thread and leave a dummy element at the position of the expired thread in the stack. If this position is the bottom, remove all consecutive dummy elements from the bottom of the stack (such that the deepest non-dummy idle thread becomes the new bottom.)

(3) When a new job arrives, remove the top element from the stack. If it is an idle thread then assign the job to it. However if it is a dummy element then create a new thread for the job.

It may seem stiff and contrary to the goals that we create a new thread if the top element is dummy, although other idle threads may still exist. However the point is that we easily get a proof of $r$-competitiveness. Afterwards we present a more natural version of the algorithm that renounces dummy elements at all and assigns every new job to the topmost idle thread, unless the stack is empty. The $r$-competitiveness argument becomes more complicated, thus we found it smoother to insert the preliminary Version 1 and do the proof by comparison.

**Theorem 4.2** *EXPIRATION DATE STACK (R), Version 1, is $r$-competitive.*

**Proof.** Using our framework from Section 2, we see that this algorithm is specified by $s(P) = \min\{b, w(P)\}$, with $b$ chosen according to $R$, independently for all down-up pairs $P$ of $f$. Now $r$-competitiveness follows similarly as in Theorem 3.1:

For each $P$, the algorithm pays $w(P)$ if $w(P) \leq b$, and $1 + b$ if $w(P) > b$. The assumption that $R$ is an $r$-competitive rent-to-buy algorithm means that the expected value of this cost is at most $r$ times the optimum cost $\min\{1, w(P)\}$, for any value $w(P)$. By the decomposition explained in Section 2 and linearity of expectation, the total expected cost is within factor $r$ of optimum. □

**EXPIRATION DATE STACK ($R$), Version 2**

Maintain a stack of threads.
(1) When a thread becomes idle at time $t$, assign expiration date $t + b$ to it, where $b$ is chosen according to $R$, and add it to the stack.
(2) When the expiration date of a thread is reached, delete this thread and remove it from the stack. (The stack may be implemented as a doubly linked list, such that removal is fast.)
(3) When a new job arrives, remove the top element from the stack and assign the job to it.

**Theorem 4.3** *EXPIRATION DATE STACK (R), Version 2, is $r$-competitive.*

**Proof.** By Theorem 4.2 it suffices to show that, for every $f$ and every random string of $b$'s, Version 2 is not more expensive than Version 1. Consider any down-up pair $P$ of $f$. At the upwards step of $P$, Version 1 uses the thread that became idle at the downwards step of $P$, if it still exists, otherwise it creates a new one. In the following we call it "the thread of $P$", for convenience.

From the rules of both algorithms, it is not hard to see that the set of down-up pairs of $f$ can be partitioned into chains, such that in every chain $(P_1, \ldots, P_k)$ with $k > 1$ the following holds:

- $P_{i+1}$ has a higher ordinate than $P_i$.
- The thread of $P_1$ expires before the upwards step of $P_1$.
- For $1 \leq i < k$, at the upwards step of $P_i$, the thread of $P_{i+1}$ becomes busy.
- At the upwards step of $P_k$, a new thread is created.

Note that, within every chain, the sum of idle times in Version 2 is at most the sum of idle times in Version 1 (since threads become busy earlier). Furthermore, Version 2 creates only one new thread in every chain, whereas Version 1 creates at least one new thread (in $P_1$). □

An important issue in all randomized algorithms is the amount of randomness. First of all, if $R$ itself is a continuous algorithm (such as KMMO), then, in order to devote constantly many random bits to each $b$, we must use a discretized version of $R$ with slightly larger competitive ratio instead. Thus the following discussion refers to discrete distributions $R$ only.

EXPIRATION DATE STACK ($R$) is highly random if we apply $R$ independently to all threads that become idle. On the other hand, the proof of $r$-competitiveness does not rely on independent applications of $R$ at all. (Linearity of expectation holds for arbitrary random variables.) So we might even go to the other extreme and fix a single $b$ according to $R$ which is then used in all (!) down-up pairs. The obvious drawback is that certain workload functions can foil such a solution, i.e. produce an actual competitive ratio significantly larger than $r$. (For example, consider a 0,1-valued $f$ where the $f = 0$ intervals have length $b + \varepsilon$.) In contrast, independent applications of $R$ make the competitive ratio sharply concentrated around $r$ for any large enough input.

It arises the question whether we can achieve a competitive ratio sharply concentrated around $r$ on any input, while using less randomness. In the next sections we propose such a version of EXPIRATION DATE STACK ($R$). Regardless the size of $f$ it uses a constant amount of randomness per time, however, the expected competitive ratio is slightly larger than the optimum $e/(e-1)$, with an excess depending on this amount. In principle, this is the best one can get, since some trade-off between randomness and $r - e/(e-1)$ is inevitable already for rent-to-buy.

# 5 Discrete Randomized Rent-to-Buy

As a preparation, we return to the single rent-to-buy problem.

**Definition 5.1** *A discrete rent-to-buy algorithm $R$ with denominator $n$ is a probability distribution on a set of $n$ points $0 \le t_1 \le \ldots \le t_n$ in which $R$ buys at each time $t_k$ with probability $1/n$. Let $r$ denote the expected competitive ratio of $R$.*

The problem is to find points $t_k$ so as to minimize $r$, for fixed denominator $n$. This is also interesting for its own, since a discrete strategy does not need real computations, unlike the continuous KMMO algorithm. The $t_k$ may be computed once and stored in a table. Note that our problem is "orthogonal" to that of randomized snoopy caching solved in [6] where $t_k = k/n$ are fixed and the probabilities are the variables.

In an instance of the rent-to-buy problem, let $t$ denote the duration for which the resource is needed. ($t$ is unknown to the online player). Remember that $\min\{t, 1\}$ is the optimum cost. For convenience we additionally define $t_0 = 0$ and $t_{n+1} = \infty$. We express $r$ by

$$r = \max_{0 \le k \le n} r_k$$

where $r_k$ is the worst-case expected competitive ratio, provided that $t_k \le t < t_{k+1}$. Next we study $r_k$, that is, we assume $t_k \le t < t_{k+1}$. Define

$$s_k = \sum_{i=1}^{k} t_i.$$

The expected cost incurred by $R$ is $n^{-1}(s_k + k + (n-k)t)$, since we pay $1 + t_k$ if we buy at point $t_k$, and we pay $t$ if we didn't buy until time $t$. We distinguish three cases:

- If $t_{k+1} \le 1$ then $r_k$ is the cost divided by $t$, which is maximized if $t = t_k$. Hence we have $r_k = n^{-1}(\frac{s_k+k}{t_k} + n - k)$ in this case, for $k \ge 1$, and $r_k = 1$ for $k = 0$.

- If $1 < t_k$ then $r_k$ is the cost of $R$, which is maximized if $t = t_{k+1}$, hence $r_k = n^{-1}(s_k + k + (n-k)t_{k+1})$. In case $k = n$ we get $r_n = n^{-1}(s_n + n)$.
- If $t_k \leq 1 < t_{k+1}$ then $r_k$ is the maximum of both expressions mentioned above.

Now we can prove:

**Lemma 5.2** *In an optimal $R$ with denominator $n$ we have $t_n = 1$ and $r = \max_{1 \leq k \leq n} r_k$, where*

$$r_k = n^{-1}\left(\frac{s_k + k}{t_k} + n - k\right).$$

**Proof.** Assume that $t_n > 1$. Let $u$ be that index with $t_u \leq 1 < t_{u+1}$. By the above discussion, the $t_k$, $k > u$, do not appear in the $r_k$ expressions with $k \leq u$. Furthermore, every $r_k$, $k > u$, is a monotone increasing function in all arguments $t_i$. It follows that for any fixed vector $t_1, \ldots, t_u$ we get minimum $r$ if each $t_k$, $k > u$, equals $t_u$, instead of having the given value. This shows $t_n \leq 1$. We conclude that $r_k = n^{-1}(\frac{s_k+k}{t_k} + n - k)$ holds for all $k \geq 1$. The term $r_0 = 1$ is redundant.

Now assume $t_n < 1$. If we multiply all $t_i$ by the same factor $a > 1$ then all $r_k$ decrease. Hence $r$ is minimized if we choose $a$ possibly large, but this means $t_n = 1$. □

Since $r_k$ is monotone decreasing in $t_k$ and monotone increasing in all $t_i$, $i < k$, we get minimum $r$ if $r_1 = \ldots = r_n$. So Lemma 5.2 yields $n-1$ algebraic equations in $n-1$ variables $t_k$, $k < n$. We illustrate case $n = 2$:

**Corollary 5.3** *The best $R$ with denominator 2, given by $t_1 = (\sqrt{5} - 1)/2 \approx .62$, has competitive ratio $(5 + \sqrt{5})/4 \approx 1.81$.*

**Proof.** By Lemma 5.2, $r_1 = 1 + 1/2t_1$ and $r_2 = 1 + (t_1 + 1)/2$. Thus $t_1^2 + t_1 - 1 = 0$. □

Denominators $n > 2$ lead to higher-order algebraic equations which do not seem to have "nice" solutions. However they may be solved numerically. For $n = 3$ we get $t_1 \approx .45$, $t_2 \approx .78$, and $r \approx 1.75$, etc. In general, we can at least show that $r \sim e/(e-1) + 1/2n$, subject to lower-order terms:

**Theorem 5.4**

$$\frac{e}{e-1} + \frac{1}{2n} - \frac{1}{n^2} < r < \frac{e}{e-1} + \frac{1}{2n}.$$

**Proof.** Recall from the proof of Lemma 5.2 that any $R$ with $t_n \leq 1$ has the following properties: $r = \max_{1 \leq k \leq n} r_k$ where $r_k = n^{-1}(\frac{s_k+k}{t_k} + n - k)$, and the worst-case ratio of the expected cost to $t$ occurs at some $t = t_k$.

We show that just a discretization of KMMO (rather than an optimal $R$) gives our upper bound. Namely, choose $R$ with $t_k = \ln(1 + (e-1)k/n)$, in particular $t_n = 1$. Compare the expected costs of $R$ and KMMO at any fixed $t = t_k$. By our choice of $t_k$, the probability to buy until $t_k$ is $k/n$ in both algorithms. It remains to estimate the difference $d_k$ of the expected rent times of $R$ and KMMO.

The probability to buy the resource in interval $(t_i, t_{i+1}]$ is $1/n$ in both algorithms, but $R$ defers the buy decisions of KMMO until $t_{i+1}$. Since the density function $e^t/(e-1)$ used in KMMO is monotone increasing, the average delay is at most the half interval length. Due to this observation, for $0 \leq i < k$, the contribution of interval $(t_i, t_{i+1}]$ to $d_k$ is at most $\frac{t_{i+1}-t_i}{2n}$. Since these summands telescope, we have $d_k \leq \frac{t_k}{2n}$. Finally, in $r_k$ the costs are divided by $t_k$, thus the upper bound follows.

For the lower bound, consider any discrete rent-to-buy algorithm $R'$, specified by points $t'_1, \ldots, t'_n$. By Lemma 5.2 we may w.l.o.g. assume $t'_n = 1$. Let $k \geq 1$ be the first index with $t'_k \leq t_k$. Clearly, $r' \geq r'_k \geq r_k$. Hence it suffices to prove that each $r_k$ of our particular $R$ is larger than the asserted lower bound.

Similarly as above, we only have to estimate $d_k$, and we exploit the monotonicity and convexity of the KMMO density function. Within interval $(t_i, t_{i+1}]$, the area below the graph of this function is at most $\frac{e^{t_{i+1}}+e^{t_i}}{2} \frac{t_{i+1}-t_i}{e-1}$. The rectangle of height $\frac{e^{t_i}}{e-1}$ included therein has area $e^{t_i} \frac{t_{i+1}-t_i}{e-1}$. Thus at least a $\frac{2e^{t_i}}{e^{t_{i+1}}+e^{t_i}}$ fraction of the probability mass in this interval has an average delay of the half interval length, if buy decisions are deferred to $t_{i+1}$. From $e^{t_i} = 1 + (e-1)i/n$ we see that this expression is minimized with $i = 0$, and it simplifies to $\frac{2n}{2n+e-1}$. It follows that $d_k \geq \frac{1}{2n+e-1}$. Verify that this is at least $\frac{1}{2n} - \frac{1}{n^2}$. □

# 6 Barely Random Multiple Spin-Block Algorithms

In this section let $R$ be a fixed discrete rent-to-buy algorithm with denominator $n$ and $t_n = 1$. A useful observation is that $r_1 = 1/nt_1 + 1 \leq r$ yields $t_1 > 1/n(r-1)$.

As announced, we now provide a version of EXPIRATION DATE STACK ($R$), called Version 3, which is barely random (in the sense as defined in the Introduction) and achieves competitive ratio close to $r$ with high probability, on each workload function. For this we take Version 1 and modify Step (1) as described below. (An informal

description should be enough.)

We use an arbitrary but fixed permutation $\pi$ of the first $n$ positive integers. Define $h = t_1$. Divide the time axis by lattice points into slices of length $h$. For a lattice point $l$, let $T(l)$ be the set of threads that became idle between $l - h$ and $l$, and have not been occupied again by new jobs by time $l$. $T(l)$ is some topmost segment of the stack. Note that $R$ would not delete any job from $T(l)$ before $l$, since it has been idle for less than $t_1$ time units. So it suffices to fix the expiration dates of all idle threads in $T(l)$ only at time $l$. This is done in the following way: Put the threads of $T(l)$ in a round robin fashion into $n$ subsets, in the ordering they became idle, hence any $n$ consecutive threads from $T(l)$ get into different subsets. Sample a random integer $x \in \{1, \ldots, n\}$. Assign idle time $v = t_{(\pi(k)+x) \bmod n+1}$ to all threads in the $k$-th subset. (The expiration date of a thread is $u + v$ when it became idle at $u$.)

**Corollary 6.1** *EXPIRATION DATE STACK (R), Version 3, is r-competitive.*

**Proof.** This follows as in Theorem 4.2. Just note that, due to the random shift by $x$, every idle thread in each $T(l)$ is deleted according to distribution $R$. □

We remark that, by an obvious modification, we may also assign the same idle times to the threads as soon as they become idle: Follow $\pi$ forward/backward when a thread becomes idle/busy, and perform a random shift every $h$ time units.

Version 3 uses only a fixed number of independent random integers per time unit, regardless of the workload function $f$. Most pleasantly, $r$ is not only the expected competitive ratio. We can give a stronger guarantee. Remember that the cost incurred by a function $g \geq f$ can be considered as the sum of certain costs charged to the down-up pairs of $f$, as introduced in Definition 2.3.

**Proposition 6.2** *In EXPIRATION DATE STACK (R), Version 3, we have: The cost of all but constantly many down-up pairs of $f$ in every $T(l)$ is surely at most $r$ times the optimum.*

**Proof.** As in Version 1, every thread is assigned to a fixed down-up pair of $f$, as soon as it becomes idle. Partition $T(l)$ into subsets $T(l)_k$ $(0 \leq k \leq n)$, being the sets of threads in $T(l)$ which are responsible for down-up pairs $P$ of $f$ with $t_k \leq w(P) < t_{k+1}$. Obviously, every $T(l)_k$ is a contiguous subset of elements on the stack, at time $l$. Further partition each $T(l)_k$ into contiguous blocks of exactly $n$ threads. There remain

less than $n$ threads in each $T(l)_k$ which are not in these blocks, and hence less than $n^2$ such threads in $T(l)$.

Due to the "modulo construction", every block contains exactly one thread with idle time $t_j$, for $1 \leq j \leq n$. This fact together with the formula in Lemma 5.2 means that the cost of each block in any $T(l)_k$ is at most factor $r_k$ away from optimum. Finally note $r_k \leq r$. □

A straightforward consequence is a bound on the standard deviation for any workload function $f$ with many "broad" down-up pairs being optimal subject to constant factors.

**Theorem 6.3** *In EXPIRATION DATE STACK (R), Version 3, with fixed R (and n), the standard deviation of the competitive ratio is less than $O(1/\sqrt{p})$ on any workload function having p down-up pairs of width at least h.*

**Proof.** In every $T(l)$, the excess of costs above $r$ times the optimum is bounded by the number of threads outside blocks. These are less than $n^2$ threads. Hence the variance of these extra costs in $T(l)$ is constantly bounded. Since independent random shifts are used in different $T(l)$, the variance of total extra costs is easily seen to be $O(p)$, and the standard deviation is $O(\sqrt{p})$. Since each of the $p$ down-up pairs has at least the constant optimum cost $h$, the standard deviation of the competitive ratio behaves as claimed. □

Note that narrow down-up pairs (of width less than $h$) are optimally served by the algorithm. To see that $O(1/\sqrt{p})$ is the best standard deviation one can guarantee, consider functions $f$ jumping between 0 and 1 only, with down-up pairs of width between $h$ and 1. Then the only thread is deleted at most $p$ times independently, so that the extra costs follow a binomial distribution with standard deviation $O(\sqrt{p})$, whereas the cost is $\Theta(p)$.

The results in this section work for any fixed cyclic ordering $\pi$ of the $t_k$ values. Thus we may use a particular $\pi$ where the $t_k$ are "well mixed" in the following sense: For each $k$ and $m$, all segments of length $m$ in the cyclic ordering contain "almost the same" number of $t_i$ values with $i \leq k$. Remember that, in each $T(l)_k$ in Proposition 6.2, the threads outside blocks form a contiguous set, hence we conjecture that such $\pi$ yields the smallest standard deviation. However we abstain from an analysis of this rather marginal subject. The following is a good choice: Let $\phi = (3-\sqrt{5})/2$ (thus $1-\phi$

is the golden ratio). Define a total ordering $\prec$ by $t_i \prec t_j$ iff $i\phi - \lfloor i\phi \rfloor < j\phi - \lfloor j\phi \rfloor$, and use the cyclic ordering obtained from $\prec$.

In analogy to Version 2 we may finally introduce Version 4 where the stack contains no dummy elements, i.e. expired threads are removed immediately from the stack, also from inner positions. As in Theorem 4.3, Version 4 is not more expensive than Version 3, for any worklaod function and any sequence of random integers.

However, in Version 4 it becomes crucial to a real implementation how we handle the deletion of expired threads. Using a timer for each thread is out of the question. It is too expensive and may even cause a server crash [2]. Instead we use $n$ timers (i.e. a constant number!), one for each set $T_k$ of idle threads whose idle time is fixed to be $t_k$. The $k$-th timer notifies the program if the earliest expiration date in $T_k$ is reached. Then it deletes one thread and sets the alarm for the next expiration date in $T_k$ which is found in $n$ expected steps: Since the $t_k$ are drawn from a cyclic ordering, interrupted by random shifts, we may simply search the stack bottom-up, until we meet a thread from $T_k$. (This is not true in Version 2, where we must maintain an auxiliary data structure in order to quickly find the next expiration date!) In practice, the choice of $n$ is a compromise between competitive ratio (cf. Theorem 5.4) and administration overhead.

# 7 Some Extensions

We briefly report two extensions of our study. The proofs are left to the reader. No arguments than those given in earlier sections are required.

We have seen that a suitable algorithm with lookahead 1 gives optimal cost and studied online algorithms with lookahead 0. It is natural to ask what is the competitive ratio for lookahead $L$ if $0 < L < 1$. An optimal deterministic rent-to-buy algorithm buys at time $1 - L$ if it notes that the resource is needed at least until time 1. We get $r = 2 - L$, and optimality is easy to see. This competive ratio applies to the multiple spin-block problem with lookahead $L$, due to the decomposition argument. Similarly as in [6], we get a randomized rent-to-buy algorithm using density function $\frac{e^{\frac{t}{1-L}}}{(e-1)(1-L)}$ for $0 \leq t \leq 1 - L$, and 0 elsewhere. The expected competitive ratio is $r = \frac{e-L}{e-1}$. Note that this is the KMMO bound if $L = 0$, and is 1 if $L = 1$. Again, this yields randomized spin-block algorithms with lookahead $L$ and the same competitive ratio.

We have assumed that the costs of create/delete operations and running idle threads are accurately known. The choice of the time unit depends on this knowledge. In

practice we must use an estimation, obtained from experiments. So assume that we apply our algorithms based on a wrong time unit $U \neq 1$. Here $U < 1$ ($U > 1$) means that the create/delete costs are underestimated (overestimated.) We get $r = 1 + 1/U$ if $U < 1$, and $r = 1 + U$ if $U > 1$. The randomized KMMO algorithm would use the density function $\frac{e^{t/U}}{(e-1)U}$ for $0 \leq t \leq U$. Some case distinctions and tedious calculations give $r = \frac{1}{e-1} + \frac{1}{U}$ if $U < 1$, and $r = 1 + \frac{U}{e-1}$ if $U > 1$. Again, the results extend to the multiple spin-block problem. Note that our randomized algorithms give significantly better results than the simple deterministic rule, also in case of badly estimated costs. The benefit vanishes only if $U$ is much smaller than the true value, i.e. if idle threads are deleted too rashly.

## Acknowledgment

I would like to thank the referees for a number of careful remarks.

## References

[1] A. Borodin, R. El-Yaniv: *Online Computation and Competitive Analysis*, Cambridge Univ. Press 1998

[2] T. Damaschke: private communication on a business component software project (1998-99)

[3] R. El-Yaniv, R. Kaniel, N. Linial: Competitive optimal on-line leasing, *Algorithmica* 25 (1999), 116-140

[4] D.R. Dooly, S.A. Goldman, S.D. Scott: On-line analysis of the TCP dynamic acknowledgment delay problem, *J. of the ACM* 48 (2001), 243-273

[5] J.A. Garay, I.S. Gopal, S. Kutten, Y. Mansour, M. Yung: Efficient on-line call control algorithms, *J. of Algorithms* 23 (1997), 180-194

[6] A.R. Karlin, M.S. Manasse, L.A. McGeoch, S. Owicki: Competitive randomized algorithms for non-uniform problems, *Algorithmica* 11 (1994), 542-571

[7] P. Krishnan, P.M. Long, J.S. Vitter: Adaptive disk spindown via optimal rent-to-buy in probabilistic environments, *Algorithmica* 23 (1999), 31-56

[8] D. Lea: *Concurrent Programming in JAVA. Design Principles and Patterns*, Addison-Wesley 1997

[9] R. Motwani, S. Phillips, E. Torng: Nonclairvoyant scheduling, *Theor. Comp. Sc.* 130 (1994), 17-47; extended abstract in: *4th SODA'93*, 422-431

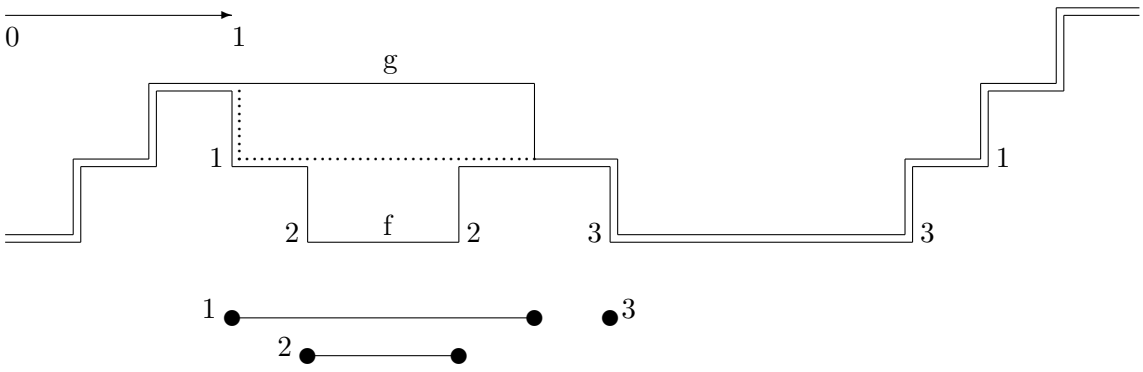


**Figure 1.** The figure illustrates a pair of functions $f$ and $g$. The time axis is horizontal, the arrow in the left upper corner has unit length. Three down-up pairs $P_i$, $i = 1, 2, 3$ of $f$ are shown, along with segments of length $s(P_i)$ given by $g$. Note that $s(P_3) = 0$. When the part of $g$ starting at $d(P_1)$ is replaced with the dashed line, we get the behaviour of BRIDGES for this workload function $f$.