

# Randomized Cup Game Algorithms Against Strong Adversaries

Michael A. Bender<sup>1</sup>

William Kuszmaul<sup>2</sup>

## Abstract

In each step of the cup game on  $n$  cups, a filler distributes up to  $1 - \varepsilon$  water among the cups, and then an emptier removes 1 unit of water from a single cup. The emptier's goal is to minimize the height of the fullest cup, also known as the backlog. The cup emptying game has found extensive applications to processor scheduling, network-switch buffer management, quality of service guarantees, and data-structure deamortization.

The greedy emptying algorithm (i.e., always remove from the fullest cup) is known to achieve backlog  $O(\log n)$  and to be the optimal deterministic algorithm. Randomized algorithms can do significantly better, achieving backlog  $O(\log \log n)$  with high probability, as long as  $\varepsilon$  is not too small. In order to achieve these improvements, the known randomized algorithms require that the filler is an oblivious adversary, unaware of which cups the emptier chooses to empty out of at each step. Such randomized guarantees are known to be impossible against fully adaptive fillers.

We show that, even when the filler is just “slightly” non-adaptive, randomized emptying algorithms can still guarantee a backlog of  $O(\log \log n)$ . In particular, we give randomized randomized algorithms against an elevated adaptive filler, which is an adaptive filler that can see the precise fills of every cup containing more than 3 units of water, but not of the cups containing less than 3 units.

## 1 Introduction

In the *cup emptying game*, there are  $n$  cups, each initially empty. In each step of the game, the *filler* distributes up to  $1 - \varepsilon$  water among the cups, where  $0 \leq \varepsilon < 1$  is the *resource-augmentation parameter*. The *emptier* then selects one cup and removes a unit of water from that cup (or all of the water in the cup, if the cup contains less than one unit). The emptier's goal is to minimize the *backlog* of the system, which is defined to be the height of the fullest cup.

The cup emptying game naturally arises in the study of processor scheduling [1, 15, 27, 29, 30], and also models problems in network-switch buffer management [5, 6, 13, 17, 18, 20, 34] and in quality of service guarantees [1, 8, 27]. The game has found numerous applications in deamortization [3, 4, 14–16, 21, 24, 33], where by considering pieces of data structures as cup emptying games that interact with one another, cup emptying algorithms can be used to spread work across many steps.

Beginning in the 1960s, much of the work on cup emptying games focused on *fixed-rate games*, in which each cup  $j$  receives the same amount of water  $f_j$  at every step [7–9, 19, 22, 26, 28–30, 32]. Baruah et al. [8] give an algorithm which achieves  $O(1)$  backlog in the fixed-rate game, and runs in polynomial time per step of the game.

In the past thirty years, researchers have turned their focus to the cup emptying game without the fixed-rate constraint. Adler et al. [1] showed that the simple *greedy emptying algorithm*, in which the emptier always removes water from the fullest cup, guarantees backlog  $O(\log n)$ . Moreover, the greedy emptying algorithm is known to be the optimal deterministic algorithm [1, 14]. Randomized algorithms, on the other hand, can achieve significantly smaller backlog [11, 15]. Bender et al. [11] showed that, as long as  $\varepsilon \geq \frac{1}{\text{polylog } n}$ , then a randomized version of the greedy emptying algorithm can achieve backlog  $O(\log \log n)$  at each step with high probability in  $n$ , which is known to be the optimal backlog guarantee for randomized algorithms [11, 15]. Subsequent work by Kuszmaul [25] extends these guarantees to arbitrarily small  $\varepsilon$  (even  $\varepsilon = 0$ ) for games whose lengths are polynomially bounded.

The power of randomization comes from the fact that the filler is required to be an *oblivious adversary*, meaning that it must decide at the beginning of the game how much water to put into each cup at each step. Randomness in the emptying algorithm can therefore be used to add unpredictability to the emptier's behavior, making it difficult for the filler to anticipate how much water is in each cup at a given step. The best known randomized algorithms against an oblivious adversary work by adding small “random perturbations” to the behavior of the standard greedy algorithm.

If, instead, the filler is an adaptive adversary, then no randomized emptying algorithm can achieve backlog  $o(\log n)$  [11, 14, 15]. Rather, the filler can achieve backlog  $\Theta(\log n)$  with the following simple  $(n-1)$ -step construction. In the first step, the filler spreads 1 unit of water equally among the  $n$  cups. In the next step, the filler distributes 1 unit of water equally among the  $n-1$  cups that have not yet been emptied. In each subsequent step  $t$ , the filler spreads 1 unit of water equally among  $n-t+1$  cups that have not yet been touched by the emptier. After the  $(n-1)$ -th step, one

<sup>1</sup>Stony Brook University

<sup>2</sup>Massachusetts Institute of Technology

cup remains with  $1/n + 1/(n-1) + \dots + 1/2 = \Theta(\log n)$  water. The fact that the filler is an adaptive adversary enables the filler to determine after each step which cups have not yet been touched by the emptier.

**Cup games with cyclic feedback.** Modeling the filler as an oblivious adversary makes sense as long as the arrival of water across steps is independent of the emptying decisions in previous steps. This is not true in many natural applications of cup games, however. Often, the work that is cleared from buffers on one step dictates which work arrives into buffers in subsequent steps. For example, in scheduling/queuing applications, the threads that are scheduled on a given step determine the tasks that are added to the queues on the next step. Similarly, in data-structure applications, where cup games schedule tasks within a data structure, the same type of cyclic feedback can occur [10].

As soon as there is cyclic feedback within the system, the filler can no longer be modeled as oblivious. This means that the randomized algorithms for the cup game [11, 15, 25] cannot be applied directly to the vast majority of the cup game's applications [1, 5, 6, 10, 13, 15, 17, 18, 20, 27, 29, 30, 34]. In this paper we ask the following question: When a cup game exhibits cyclic feedback, is the “correct” backlog  $O(\log n)$ , as with adaptive adversaries, or  $O(\log \log n)$ , as with oblivious adversaries, or somewhere in-between?

In general, systems that exhibit cyclic feedback can often behave poorly under stress, even when the cyclic feedback is not *explicitly* malicious. A classic example of this phenomenon is *cache stampedes* which occur in caches on distributed databases: when the lease on a frequently accessed element of a cache expires, multiple requests to that item may trigger cache misses and start repopulating that item at the same time—by bogging down the system, this can then cause additional cache items to expire, resulting in a cyclic positive feedback loop [2, 36]. Another example of this phenomenon is the *double-disk failure problem*: when data is stored redundantly on two drives, the two drives often fail at close to the same time—this is because (a) the two drives experience similar access patterns over their lifetime, and because (b) the failure of one drive implicitly causes a sudden unusual amount of activity on the other (due to rebuild efforts), thereby increasing the probability of failure on the second drive [23, 35].

When a system is under stress, the cyclic feedback between components of the system can have critical effects on the system's behavior. For a cup game, a cup is “under stress” when it has significantly more water than average. It is natural to assume that the filler is aware of which cups are under stress and of how much water is in those cups.

**An adaptive adversary with a view of stressed cups.** In this paper, we give the adversary full adaptive knowledge about all cups that are under stress. We also define “under

stress” liberally, meaning that any cup containing even a small constant factor more than average fill (in fact, fill 3), has its state revealed to the filler.

Formally, we define an *elevated adaptive filler* to be an adversary that has access to the fills of every cup, except for the cups containing 3 units of water or less. One can think of this adversary as being separated from the cups by an opaque wall of height 3 (see Figure 1). Whenever a cup contains more than 3 units of water, the cup is said to be *visible* to the elevated adaptive filler, and whenever a cup contains less than 3 units of water, the cup becomes *invisible* to the elevated adaptive filler (although the filler may still be able to infer information about the cup's fill). If a cup becomes visible during a step (because of the water placed in it by the filler), and then becomes invisible again during the same step (because the emptier empties out of it), the elevated adaptive filler is able to see the fill of the cup in-between the two actions.

Before continuing with the exposition, we note that the elevated adaptive filler is a remarkably powerful adversary in the context of trying to understand how a filler can react to parts of the system being stressed. Indeed, if our only goal were to tell the filler roughly how stressed out each individual cup actually is, then it would suffice to give the filler access to the *approximate* fill of each cup (i.e.,  $\lfloor \log f \rfloor$ ). The elevated adaptive filler is given substantially more information than this, however.

A priori it is unclear whether  $O(\log \log n)$  backlog should be achievable against an elevated adaptive adversary. On one hand, the known strategies for the filler that achieve backlog  $\Omega(\log n)$  all require detailed knowledge of the exact fills of cups [1, 11, 15], including the cups that are nearly empty, meaning that these lower bounds cannot be applied directly by the elevated adaptive filler. On the other hand, the best known emptying algorithms (namely, the smoothed greedy emptying algorithm of [11]) can easily be exploited by the elevated adaptive filler in order to reveal the random bits being used by the emptying algorithm, thereby reducing the problem to the fully oblivious case.

If a randomized algorithm is to work against an elevated adaptive adversary, then a careful balance must be achieved. On one hand, the algorithm must use randomness in order to make its behavior unpredictable to the filler. On the other hand, the algorithm must be robust to the fact that the filler has almost complete information about the state of the system at all times.

**Our results.** We show that randomized emptying algorithms can be made effective against elevated adaptive adversaries, achieving the same bound on backlog as if the adversary were oblivious. In particular, we prove that, if the filler is an elevated adaptive adversary, and the emptier has resource augmentation  $\varepsilon \geq \frac{1}{\text{polylog } n}$ , then there is a randomized

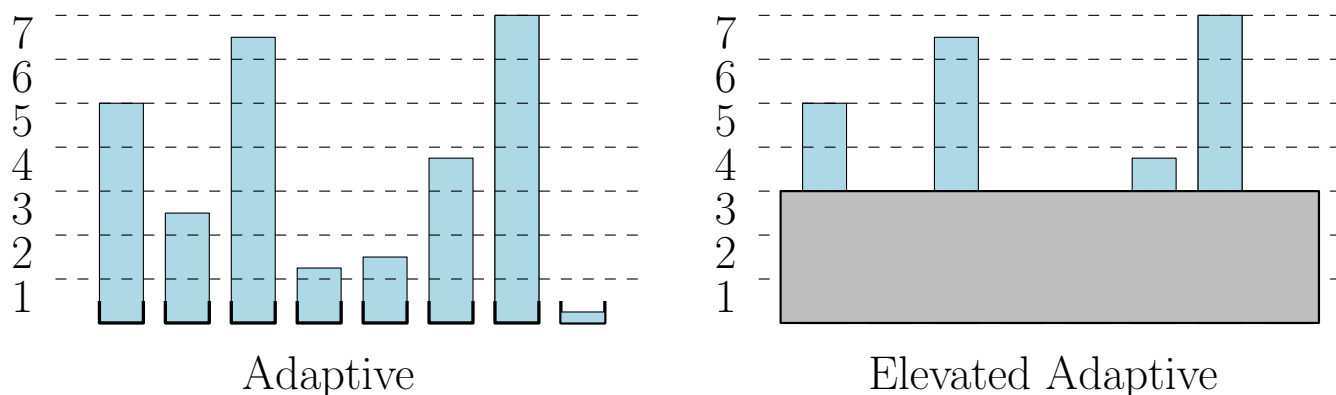


Figure 1: The same cup configuration from the perspective of an adaptive filler and from the perspective of an elevated adaptive filler.

emptying algorithm that, at any step  $t$ , achieves  $O(\log \log n)$  backlog with high probability in  $n$ . This guarantee continues to hold for each step  $t$ , even when  $t$  is arbitrarily large.

The algorithm introduced in this paper, which we call the **Dual-Mode Algorithm**, dynamically toggles between a “greedy mode” and a “prescheduled mode”. These two modes handle two extremes for how the filler could try to cause high backlog. Greedy mode aggressively removes water from the fullest cups, preventing the filler from achieving a high backlog over a small (polylogarithmic) number of steps. Prescheduled mode, on the other hand, is designed to leak water out of cups slowly in a manner that is obscured from the filler, preventing the filler from achieving high backlog over a large stretch of time. Remarkably, even though each emptying mode is designed to handle an extreme behavior by the filler, the two modes can be superimposed to achieve guarantees against arbitrary behavior from an elevated adaptive filler.

The elevated adaptive adversary does not necessarily apply out-of-the box to every application of cup games. For example, in scheduling applications, the decision of which cup is emptied affects which cups receive water in the next step, even if the emptied cup was under very low stress. On the other hand, the Dual-Mode emptying algorithm only removes water from cups that are under very low stress when the state of the system as a whole is good (i.e., all cups are under very low stress). This means that the elevated adaptive adversary *does* eliminate the possibility of a feedback loop within the system causing the system to perpetually travel towards a worse and worse state in terms of its backlog. That is, the only way that a filler can successfully behave maliciously is if that malicious behavior occurs when all of the cups in the system are under low stress.

**An application to data-structure design.** The analysis of the deterministic greedy emptying algorithm [1, 14], which appeared in its earliest form nearly thirty years ago, has found extensive applications in data-structure deamortiza-

tion [3, 4, 14–16, 21, 24, 33]. The known randomized algorithms for the cup emptying game are much more recent [11, 15], with the first algorithms to work with low resource augmentation having been introduced in 2019 [11]. Like their deterministic predecessor, these algorithms seem promising as deamortization tools.

The issue of cyclic dependencies within a data structure significantly complicates the task of applying randomized cup-emptying within a data structure, however [10]. In particular, because different parts of the data structure interact with each other, often cyclically, the cup-emptying games within the data structure are *not* against oblivious adversaries. The output of a cup emptying game in one part of the data structure may later affect the input to the same game.

Recent work by Bender et al. [10] uses randomized cup emptying in order to obtain low-latency flushing strategies for  $B^\varepsilon$ -trees [10]. The work also presents a general framework for applying randomized cup-emptying to external-memory problems in a way that exploits the presence of a small in-memory cache in order to make the “effective backlog” of each cup game constant [10]. However, the authors note that in order to apply randomized cup-emptying to any given data structure, all of the cyclic dependencies within that data structure must first be removed. Doing so for  $B^\varepsilon$ -trees required several nontrivial data-structure-specific techniques for isolating the games from one-another [10].

The elevated adaptive filler provides a more general approach to handling cyclic dependencies within a data structure: rather than *eliminating* those dependencies, we simply navigate them in a way that allows for the elevated adaptive filler to absorb them. This allows for us to get all of the bounds that we want for the cup game (i.e., backlog  $O(\log \log n)$  and a bound on the number of badly behaving cups) without having to worry about cyclic dependencies. In Section 7, we show how to apply the elevated adaptive filler to  $B^\varepsilon$ -trees, eliminating much of the data-structural overhead

introduced in [10].

**Paper outline.** We begin in Section 2 by presenting the Dual-Mode Emptying Algorithm. As described above, the Dual-Mode Algorithm alternates between two types of behaviors, depending on the recent behavior of the filler. The default mode of the Dual-Mode Algorithm is to schedule cups for emptying based on when each cup's fill crosses certain randomly assigned thresholds. This prevents cups from gradually becoming full over many steps. On the other hand, if the filler places a large amount of water into a cup over a small number of steps, then the emptier swaps into "greedy mode", and aggressively targets those cups.

By analyzing a variant of the Dual-Mode Algorithm, we prove the following theorem.

**THEOREM 1.1.** *Suppose  $\varepsilon \geq \frac{1}{\text{polylog}(n)}$  and let  $p = \frac{1}{\text{poly}(n)}$ . Consider a cup game on  $n$  cups with resource-augmentation parameter  $\varepsilon$ , where the filler is an elevated adaptive adversary. Then there exists an emptying algorithm such that after each step  $t$  the backlog is  $O(\log \log n)$  with probability at least  $1 - p$ .*

In Section 3, we give a technical overview of the analysis, and motivate the key ideas in the design of the Dual-Mode Algorithm. The analysis of the algorithm is then spread across three sections.

In Section 4, we show that the only way for a filler to cause the Dual-Mode Algorithm to exhibit large backlog, is if the filler tricks the algorithm into making a large number of bad scheduling decisions in a short time period. When this happens, the emptier exhibits a failure mode in which it halts.

In Section 5 we bound the probability that the Dual-Mode Algorithm is forced to halt. To do this, we first show that the Dual-Mode Algorithm renders the elevated adaptive filler no more powerful than a simpler type of adversary that we call an "integer-updated filler". Then we prove that the integer-updated filler is unable to effectively distinguish between the Dual-Mode Emptying Algorithm and a version of the emptying algorithm that is augmented with certain "superpowers" that allow the emptier in certain cases to change the past retroactively. Finally, we prove that these superpowers prevent the filler from learning any information that might allow for it to successfully manipulate the emptier into halting.

Section 6 puts together the results from the preceding sections to complete the proof of Theorem 1.1. In order to achieve guarantees for arbitrarily long games, the section introduces an additional "recovery mode" to the algorithm that is invoked whenever the algorithm halts.

Finally in Section 7 we discuss an application of the Dual-Mode Emptying Algorithm to external memory data structures.

## 2 The Dual-Mode Emptying Algorithm

**2.1 Algorithm overview** The structure of a round. The Dual-Mode Emptying Algorithm operates in **rounds**. During each round  $r$ , there are a fixed number  $\ell = \text{polylog}(n)$  of steps during which the emptying algorithm is in **prescheduled mode**, and some variable number (possibly 0) of steps during which the algorithm is in **greedy mode**. During each round  $r$ , cups can be **scheduled** to be emptied during round  $r + 1$ . Each cup  $j$  that is scheduled to be emptied during round  $r + 1$  (actually, the same cup can be scheduled up to two times per round) is assigned a random  $i \in \{1, 2, \dots, \ell\}$ , and the emptier removes 1 unit of water from cup  $j$  during the  $i$ -th prescheduled-mode step of round  $r + 1$ . (Note that this is distinct from the  $i$ -th step of round  $r + 1$  due to greedy-mode steps.) If a prescheduled step  $i$  does not have any emptyings assigned to it, then the emptier simply skips its turn during the  $i$ -th prescheduled step. A round is defined to terminate after its  $\ell$ -th prescheduled step occurs.

Whenever the emptier removes water from a cup  $j$  (in either greedy or prescheduled mode), the algorithm always removes a full 1 unit of water. In particular, the design of the Dual-Mode Algorithm ends up ensuring that the emptier only ever removes water from cups that contain at least one full unit of water (see Lemma 4.1).

We remark that, although the length of a round can vary based on the number of greedy-mode steps, it turns out that the length will never exceed  $\text{polylog } n$  (see Lemma 4.3).

**Toggleing between two modes.** Whenever a cup receives more than  $O(1)$  units of water in a single round, the cup is said to require **greedy emptying**, and the algorithm swaps into greedy mode. When in greedy mode, the algorithm empties 1 unit of water from the fullest cup that requires greedy emptying.

The idea is that, if the adversary is hammering on a small (polylogarithmic) number of cups during a round, then we cannot afford to delay emptying from those cups; in this case greedy mode performs well because the filler cannot get a large backlog (against the greedy emptying algorithm) by focusing on a small number of cups.

If, on the other hand, the filler is distributing water gradually into cups, then we wish to schedule the cups for emptying in a way that obfuscates our decisions from the filler; we can do that with prescheduled mode. Because prescheduled mode deals with the slow arrival of water, the mode can afford to make scheduling decisions that do not take effect until the next round (i.e., for polylogarithmically many steps). This allows the mode to randomize its scheduling decisions in a way that is difficult for the elevated adaptive filler to exploit.

The two modes of the algorithm are each equipped to handle one extreme of how the filler might behave (either slowly placing water into many cups, or quickly placing

water into just a few cups). The interaction between the two modes allows for them also to handle any complex filling behavior that lies between these two extremes.

**Scheduling with random thresholds and deciding when to empty greedily.** The decision in round  $r$  of which cups are scheduled to be emptied from during round  $r + 1$  is made using random thresholds in each cup. Roughly speaking, each time that the fill of a cup  $j$  increases from some integer  $k$  to some integer  $k + 1$ , the cup crosses a randomly selected threshold in  $[k, k + 1)$ .<sup>1</sup> The scheduling of cup  $j$  occurs in the round  $r$  in which the random threshold is crossed, and cup  $j$  is then emptied out of during a random prescheduled-mode step in the next round  $r + 1$ .

There is an important caveat to this rule that pushes us into greedy mode: If the fill of cup  $j$  increases from one integer  $k$  to another  $k + 1$  all within a *single* round  $r$ , then the random threshold between  $k$  and  $k + 1$  is not considered to have been crossed, and instead the cup is designated as requiring 1 unit of greedy emptying. In this case, if the algorithm is not already in greedy mode, it enters greedy mode.

**A failure mode.** If the Dual-Mode Algorithm schedules more than  $\ell$  cup emptyings to be performed during a round  $r$ , then the algorithm enters a failure mode in which it halts. On the other hand, until the algorithm halts, the backlog is guaranteed not to exceed  $O(\log \log n)$  (we will prove this in Section 4). The algorithm uses carefully designed randomness in its scheduling decisions in order to ensure that the filler is unable to reliably make the algorithm halt, even if the filler is an elevated adaptive adversary.

**2.2 The two modes** Below, we describe in detail how the two modes of the Dual-Mode Algorithm operate in each round. The pseudocode for a given round is given by Algorithm 1.

**Prescheduled mode: scheduling thresholds to slots.** We begin by defining the *crossing weight* of each cup, which is a quantity that helps us keep track of when random thresholds are crossed in the cup. Define the *crossing weight*  $g_j(t)$  of a cup  $j$  after step  $t$ , to be the total amount of water placed by the filler into cup  $j$  during steps  $1, 2, \dots, t$ , minus the total number of times the emptier has removed water in greedy mode from cup  $j$  during steps  $1, 2, \dots, t$ .<sup>2</sup> Intuitively, the crossing weight is the total amount of water that has been

placed into the cup, ignoring the water that has been removed by greedy mode.

For each cup  $j$  and each  $i \in \{1, 2, 3, \dots\}$ , define the random threshold  $\tau_j(i)$  to be a random real number in  $[i, i + 1)$ .<sup>3</sup> A cup  $j$  crosses random threshold  $\tau_j(i)$  in round  $r$  if the crossing weight of the cup increases from being at most  $\tau_j(i)$  at the beginning of the round to exceeding  $\tau_j(i)$  at the end of the round; that is,  $g_j(t_0) \leq \tau_j(i) < g_j(t_1)$ , where  $t_0$  is the final step before round  $r$  and  $t_1$  is the final step in round  $r$ . Note that, if the greedy mode removes water from cup  $j$  during round  $r$ , then there may be a time  $t$  in the round at which  $g_j(t) > g_j(t_1)$ . In this case, the random thresholds  $\tau_j(i)$  satisfying  $g_j(t_1) \leq \tau_j(i) < g_j(t)$  are *not* considered to be crossed during round  $r$ .

The random thresholds  $\tau_j(i)$  that are crossed in round  $r$  are assigned to random *slots*  $k \in \{1, 2, \dots, \ell\}$  in round  $r + 1$ , where  $k$ -th slot of the round  $r + 1$  is defined to be the  $k$ -th prescheduled-mode step. If a threshold  $\tau_j(i)$  is assigned to a slot  $k$ , then this means that cup  $j$  will be emptied from during the  $k$ -th prescheduled-mode step of round  $r + 1$ . If more than  $\ell$  threshold crossings occur in a round  $r$ , then the algorithm is unable to schedule all of them to slots in round  $r + 1$ , and the algorithm halts.

**Greedy mode: preventing too many crosses in a single cup.** Greedy mode prevents any cup  $j$  from crossing more than two random thresholds during a given round. In particular, we design the algorithm to guarantee that if cup  $j$  contains crossing weight  $b_j(r)$  at the beginning of a round  $r$ , then the crossing weight at the end of the round will satisfy  $b_j(r + 1) < \lceil b_j(r) \rceil + 1$ .

This guarantee is enforced by greedy mode as follows. Whenever the crossing weight of a cup  $j$  is  $\lceil b_j(r) \rceil + 1$  or larger during a round  $r$ , the cup  $j$  is said to *require greedy emptying*. During a given step  $t$ , if any cups require greedy emptying, then the algorithm operates in greedy mode and removes 1 unit of water from the fullest cup that requires greedy emptying. When this occurs, the crossing weight of the cup is reduced by 1.

**When scheduling decisions occur.** In principle, the algorithm could wait until the end of the each round  $r$  before determining which slots to assign which thresholds to in round  $r + 1$ . For convenience of the algorithm's analysis, however, we have the algorithm instead make the scheduling decision for each threshold  $\tau_j(i)$  as soon as the algorithm can deduce that the random threshold has been crossed. We describe the precise rule for when the algorithm can make this deduction (and therefore make the scheduling decision) below.

There are two random thresholds that a cup  $j$  might

<sup>1</sup>For now, we ignore the case in which water is removed from cup  $j$  between the two integer crossings, thereby making it so that the next integer reached by the cup is not  $k + 1$  but is instead some  $k' \neq k + 1$ . This case will be considered when we describe the algorithm in more detail shortly.

<sup>2</sup>Each time that water is removed from cup  $j$ , exactly 1 unit will be removed, but since this is a nontrivial property of the algorithm, we do not assume it in the definition of  $g_j(t)$ .

<sup>3</sup>One could also include a threshold  $\tau_j(0) \in [0, 1)$ . The decision not to do so ensures that all empties performed by the emptier are always able to remove a full unit of water, however.

cross in a round  $r$ , namely  $\tau_j(\lceil b_j(r) \rceil - 1)$  and  $\tau_j(\lceil b_j(r) \rceil)$ . For example, if the crossing weight is 10.2 at the beginning of the round, then the cup can potentially cross threshold  $\tau_j(10) \in [10, 11)$  and  $\tau_j(11) \in [11, 12)$ . If cup  $j$  crosses threshold  $\tau_j(\lceil b_j(r) \rceil - 1)$ , then on the same step the Dual-Mode Algorithm schedules the threshold to a random slot in round  $r + 1$  out of the slots that have not yet been scheduled. On the other hand, if cup  $j$  crosses threshold  $\tau_j(\lceil b_j(r) \rceil)$  in round  $r$ , then there is no specific step at which the crossing definitively occurs. This is because, even if the crossing weight during some step  $t$  increases above  $\tau_j(\lceil b_j(r) \rceil)$ , the Dual-Mode Algorithm may later *reduce* the crossing weight back below  $\tau_j(\lceil b_j(r) \rceil)$  while in greedy mode. As a result, if a cup  $j$  crosses threshold  $\tau_j(\lceil b_j(r) \rceil)$ , then the Dual-Mode Algorithm waits until the end of round  $r$  to schedule the random threshold to a random remaining slot in round  $r + 1$ . If the algorithm ever tries to schedule a threshold  $\tau_j(i)$  to a slot in round  $r + 1$ , but there are not remaining slots to be scheduled to, then the algorithm halts.

### 3 Technical Overview

In this section, we give an overview of the algorithm analysis. A priori, there are two ways in which the algorithm might fail. The first is that the algorithm may allow for backlog greater than  $O(\log \log n)$ . The second is that the algorithm may halt, if the filler can force the algorithm to schedule more than  $\ell$  cups to be emptied from during a single round  $r$ . We will show that the first failure mode never occurs and that the second occurs only with a very small probability.

**3.1 Bounding the backlog** We begin the analysis by showing that the first failure mode never occurs. That is, as long as the algorithm does not halt, the backlog is guaranteed to be  $O(\log \log n)$ .

**Clean starts for each round.** A central property of the algorithm is that, at the end of each round, every cup contains at most constant fill. The mechanism for scheduling cups to be emptied ensures that, although water may gradually build up in a cup  $j$ , once that water gets to the next random threshold, the cup will be scheduled for emptying in the next round. This prevents any cup from slowly achieving a large fill.

On the other hand, if the filler attempts to quickly increase the fill of a cup  $j$ , then the emptying algorithm will enter greedy mode and ensure that most of the water has been removed from cup  $j$  by the end of the current round.

The combined effect of the two emptying modes is that when each round ends, every cup is guaranteed to be nearly empty. In order to bound the backlog, it therefore suffices to show that the greedy mode never allows the backlog to get too large within a single round.

**The setting where greedy does well.** Although the greedy

---

#### Algorithm 1 A Round of the Dual-Mode Emptying Algorithm

---

**Description:** The round begins with step  $t_0 + 1$ . The random thresholds  $\tau_j(i)$  have already been determined.

We are given a set of assignments  $\mathcal{A} \subseteq \{(i, j) \mid i \in [\ell], j \in [n]\}$  of which slots  $i$  are scheduled to empty out of which cups  $j$  during the round. We output the set of assignments  $\mathcal{B}$  used by the next round.

```

1:  $i \leftarrow 1 \triangleright i - 1$  is the number of prescheduled-mode steps so far.
2:  $t \leftarrow t_0 \quad \triangleright t$  is the current step number.
3:  $b_j \leftarrow g_j(t_0)$  for each cup  $j \triangleright b_j$  is the crossing weight of cup  $j$  prior to the round beginning.
4:  $\mathcal{B} \leftarrow \emptyset$ 
5: while  $i \leq \ell$  do
6:    $t \leftarrow t + 1$ 
7:   Filler takes a turn placing some amount  $c_j(t)$  of water into each cup  $j$ 
8:   for  $j$  s.t.  $b_j > 1$  and  $g_j(t - 1) \leq \tau_j(\lceil b_j \rceil - 1) < g_j(t - 1) + c_j(t)$  do
9:     if  $|\mathcal{B}| \geq \ell$  then Halt end if
10:    Add  $(a, j)$  to  $\mathcal{B}$  for a random  $a \in [\ell] \setminus \{a \mid (a, d) \in \mathcal{B} \text{ for some } d\}$ .
11:  end for
12:  if  $g_j(t - 1) + c_j(t) \geq \lceil b_j \rceil + 1$  for some cup  $j$  then
     $\triangleright$  Operate in greedy mode.
13:    Let  $j$  be the fullest such cup and remove 1 unit of water from cup  $j$ 
14:  else  $\triangleright$  Operate in prescheduled mode
15:    if  $(i, j) \in \mathcal{A}$  for some  $j$  then
16:      Remove 1 unit of water from cup  $j$ 
17:    end if
18:     $i \leftarrow i + 1$ 
19:  end if
20: end while
21: for cups  $j$  such that  $g_j(t) > \tau_j(\lceil b_j \rceil)$  do  $\triangleright$  These crossings can only be detected at end of round.
22:  if  $|\mathcal{B}| \geq \ell$  then Halt end if
23:  Add  $(a, j)$  to  $\mathcal{B}$  for a random  $a \in [\ell] \setminus \{a \mid (a, b) \in \mathcal{B} \text{ for some } b\}$ .
24: end for

```

---

emptying algorithm allows for backlogs as large as  $\Theta(\log n)$  in general [1], there are certain situations in which the algorithm does substantially better. Notably, the greedy emptying algorithm does very well for *short games*. If a game lasts for only  $\text{polylog}(n)$  steps, then the greedy algorithm guarantees a backlog of  $O(\log \log n)$ .

**Bounding the length of each round.** In order to show that each round has small backlog, it suffices to prove that each round has at most  $\text{polylog } n$  steps. Since the number of prescheduled steps is  $\ell \leq \text{polylog } n$ , we focus on the number of greedy-mode steps. Because of resource augmentation, each greedy-mode step removes  $\varepsilon$  more water from the system than is placed into the system during that step. Moreover, greedy-mode steps only ever remove “fresh water”, i.e., water placed into cups during the current round. The only way that greedy-mode steps can each remove  $\varepsilon$  more water than they put in is if that  $\varepsilon$  water is “supplied” by prescheduled-mode steps. In order for there to be a large number  $k$  of greedy-mode steps, it follows that at least  $\varepsilon k$  prescheduled-mode steps must occur. Since  $\varepsilon \geq \frac{1}{\text{polylog } n}$ , this bounds the number of greedy-mode steps by  $\ell/\varepsilon \leq \text{polylog } n$ .

**3.2 The probability of halting** If the filler can cause more than  $\ell$  threshold crossings to occur in a given round, then the Dual-Mode Algorithm halts. The uses of randomization in the Dual-Mode Algorithm are strategically designed to prevent the filler from causing halts, even when the filler is an elevated adaptive adversary.

**Defeating an oblivious filler.** By selecting each threshold  $\tau_j(i) \in [i, i+1)$  at random, the emptying algorithm makes it difficult for an oblivious adversary to cross many thresholds in the same round. In particular, if the amount of water placed by the filler into each cup during each step of a round is predetermined (i.e., the filler is oblivious), then the number of random thresholds crossed can then be expressed as a sum of independent 0-1 random variables with total mean at most  $(1 - \varepsilon)\ell$ , where each individual random variable indicates whether a given random threshold  $\tau_j(i)$  is crossed. It follows that with high probability, the number of threshold crossings in each round does not exceed  $\ell$ .

If the filler is not an oblivious adversary, then it may enter a round already knowing which cups are “on the verge” of crossing a random threshold. By placing water into many such cups in the same round, the filler might then force the emptying algorithm to halt. This means that, in order for the random thresholds to be effective, the filler must be kept unaware of which cups have crossed which random thresholds, until after that information stops being useful to the filler.

**Strengths and weaknesses of the elevated adaptive filler.**

Recall that the elevated adaptive filler can see the exact fill of each cup that contains more than 3 units of water; the other cups remain invisible.

One effect of this is that each cup always begins each round invisible to the filler. On the other hand, if a cup is scheduled to be emptied from during the round, and if water is placed into the cup during the round, the cup could become visible at some time before the step that water is removed from it.

In addition to seeing the fills of cups containing more than 3 units of water, the algorithm can also implicitly infer information about cups containing less than 3 units. A central goal in the analysis of the algorithm is to show that the information available to the filler cannot be used strategically to force the Dual-Mode Algorithm to halt.

There are two ways that the filler can infer information about invisible cups:

- **Local information leakage:** The fact that a cup is not visible may be sufficient information for the filler to deduce that the cup has been recently emptied from. Similarly, when a cup becomes visible, the amount of water in it can be used to determine how many times the cup was emptied from while it was invisible.
- **Inter-cup information leakage:** Any step  $t$  in which the emptier empties from some cup  $j$  is a step in which the emptier does *not* empty from any other cup  $j'$ . This simple fact can allow the filler to infer information about one cup based on information about others.

These two notions of information leakage play no formal role in the analysis of the algorithm. However, at an intuitive level, the main components of the analysis are designed to each target one of the two types of leakage. Ultimately, the analysis strategy is to reduce the game to a new game in which whenever a random decision is made by the emptier, the decision is also immediately revealed to the filler. This means that the emptier is never actually hiding information from the filler. The challenge is to perform this reduction in a way that still leaves the filler unable to use the information available to it in order to cause the emptying algorithm to halt.

**Simplifying information leakage by reducing to a different adversary.** A key idea in the analysis of the Dual-Mode Emptying Algorithm is to consider another simpler (but as we will show in the body of the paper, stronger) type of filling adversary, which we call an *integer-updated filler*. Instead of explicitly seeing the cups, the integer-updated filler receives new information (which we will specify shortly) about the cups only when a certain special event occurs, namely an *integer threshold crossing*.

An integer threshold crossing occurs in cup  $j$  whenever the total amount of water ever poured into cup  $j$  crosses some positive integer that it was previously less than. Equivalently,

an integer threshold crossing occurs whenever the crossing weight of the cup  $j$  increases to be at least some integer  $k$  that it was previously less than. We say that this integer  $k$  is the *integer threshold* that is crossed by the integer threshold crossing. Integer threshold crossings are different from random threshold crossings, but the two are related in the following way: whenever an integer threshold crossing occurs for some integer threshold  $k > 1$ , and the cup  $j$  in which it occurs does not currently require greedy emptying, then the most recent 1 unit of water to have been placed in cup  $j$  is guaranteed to have crossed the random threshold  $\tau_j(k-1)$ .

Here is the information given to the integer-updated filler whenever a cup  $j$  crosses some integer threshold  $k > 1$ . If cup  $j$  does not require greedy emptying, the filler is notified of the round and slot number to which random threshold  $\tau_j(k-1)$  is scheduled (or was scheduled in the past). Otherwise, the filler is simply told that cup  $j$  requires greedy emptying.<sup>4</sup> These two types of updates are the only information that the integer-updated filler ever receives. Nonetheless, as we show in Section 5, the integer-updated filler is at least as powerful an adversary as is the elevated adaptive filler.

A crucial property of the integer-updated filler is that, whenever it learns about the scheduling of a random threshold  $\tau_j(i)$ , sufficiently much water has already been placed in cup  $j$  that the threshold  $\tau_j(i)$  is guaranteed to be crossed. Intuitively, this means that by the time the filler learns information, it is already too late for that information to be very useful.

In some cases, the integer-updated filler may have access to information about random threshold crossings and slot scheduling before the elevated adaptive filler would have. One consequence of this is that information that the elevated adaptive filler might have learned via information leakage is often already explicitly available to the integer-updated filler, making the new adversary simpler to analyze. In particular, by restructuring when and how information is available to the filler, we completely eliminate the notion of local information leakage.

### Design decisions that limit inter-cup information leakage.

Before continuing the analysis, we first describe intuitively how the design of the Dual-Mode Algorithm eliminates inter-cup information leakage.

Recall that whenever a cup  $j$  crosses a threshold in a round  $r$ , the cup  $j$  is scheduled for emptying during a *random*

slot  $i$  in round  $r+1$  (out of the slots that have not yet been scheduled to).

The random slot assignments are designed to minimize unnecessary inter-cup information leakage. In particular, if the integer-updated filler knows about some subset of the slot assignments in some round  $r$ , the fact that those slot assignments are simply a random subset of  $[\ell]$ , prevents the filler from deducing any further information about how many other (or which other) random thresholds have been scheduled to slots in round  $r$ .

One form of inter-cup information leakage remains unavoidable. If the emptying algorithm has not yet halted, then the filler can safely deduce that at most  $\ell$  random thresholds have been scheduled to each round  $r$ . Thus every time that the filler identifies a threshold  $\tau_j(i)$  that has been scheduled to a round  $r$ , this slightly decreases the probability that each other  $\tau_{j'}(i')$  has also been scheduled to round  $r$ .

**Analyzing the Dual-Mode Algorithm by giving it superpowers.** In order to analyze the Dual-Mode Algorithm against an integer-updated filler, we define a stronger version of the Dual-Mode Algorithm that we call the Delayed-Scheduling Algorithm.

The Delayed-Scheduling Algorithm postpones deciding which slot the random threshold  $\tau_j(i)$  is scheduled to until the step at which the algorithm is forced to reveal the scheduling decision to the integer-updated filler. This means that the Delayed-Scheduling Algorithm may sometimes schedule a random threshold  $\tau_j(i)$  to a slot  $i$  in some round  $r$  that *has already occurred*.

If the filler were an elevated adaptive adversary, then the fact that the Delayed-Scheduling Algorithm can “overwrite the past” would introduce circular dependencies that prevent the algorithm from being well defined. In particular, the filler might be able to adapt to scheduling decisions that the emptier has not actually made yet. On the other hand, if the filler is integer-updated, then the only information available to the filler at a given point in time is precisely the set of scheduling decisions that the emptier has already finalized. This ensures that there cannot be circular dependencies and that the Delayed-Scheduling Algorithm is well defined. Although the Delayed-Scheduling Algorithm is impossible to implement (because it retroactively changes the past), it serves as a useful tool for analyzing the Dual-Mode algorithm.

Whereas the Dual-Mode Algorithm makes scheduling decisions for thresholds in the order that they are crossed, the Delayed-Scheduling Algorithm makes decisions for thresholds in the order that they are revealed to the filler. One important consequence is that the Delayed-Scheduling Algorithm can only halt once it has already revealed the first  $\ell+1$  thresholds that are scheduled to the offending round. This means that the filler can no longer use the fact that the algo-

<sup>4</sup>In general, any filler (even if it is oblivious) can always deduce exactly which steps are and are not in greedy mode (this is just deterministically a function of the filler’s behavior). Additionally, whenever a cup crosses an integer threshold, the filler (again, even if it is oblivious) can deduce whether the cup subsequently requires greedy emptying. Thus the only real information being received by the integer-updated filler is in the case where the cup does not require greedy emptying.

rithm has not halted yet in order to infer information about how many cups are scheduled to a round.

We use a coupling argument to prove that any integer-updated filler against the Dual-Mode Algorithm can be adapted to work against the Delayed-Scheduling Algorithm. The key insight is that, because of the randomized mechanism for assigning thresholds to slots, the two emptying algorithms are indistinguishable, up until one of them halts. That is, although the two algorithms perform scheduling decisions in different order, the act of assigning a threshold  $\tau_j(i)$  to a random slot and then another threshold  $\tau_{j'}(i')$  to a random remaining slot is equivalent to first assigning  $\tau_{j'}(i')$  to a random slot and then  $\tau_j(i)$  to a random remaining slot.

**Analyzing the Probability of Halting.** The Delayed-Scheduling Algorithm's "superpowers" make it so that the algorithm never has to hide information from the filler. By postponing decisions until right before they are revealed to the filler, the algorithm eliminates any risk that those decisions might implicitly become available to the filler via information leakage.

The final challenge in the analysis is to show that there is no strategy for the integer-updated filler that can be used to cause the Delayed-Scheduling Algorithm to halt. In the same way that the randomness in the thresholds  $\tau_j(i)$  makes it difficult for an oblivious filler to cross a large number of thresholds in a small number of steps, the randomness in the thresholds has a similar effect for the Delayed-Scheduling algorithm against an integer-updated filler. Because the integer-updated filler adapts to information as it is received, we cannot directly use the Chernoff-bound based argument that works against an oblivious filler. Instead, we complete the analysis by examining the moment generating function for (and proving a concentration inequality for) the number of integer thresholds crossed in each round, showing that with high probability the emptying algorithm does not halt.

**Achieving an Unending Guarantee.** Using the techniques described above, we prove that, with high probability in  $n$ , the Dual-Mode Algorithm does not halt during any of its first  $\text{poly } n$  steps. In order to obtain bounds on backlog for games of arbitrary (possibly super-polynomial) length, we modify the Dual-Mode Algorithm so that, at random points of time in every  $\text{poly } n$  steps, the algorithm enters a *clean-up phase* in which it greedily empties out of cups until it reaches a state in which every cup contains less than 1 unit of water. At the end of each clean-up phase, the Dual-Mode Algorithm is restarted scratch.

These periodic clean-up phases enable the algorithm to recover when it halts. On the other hand, because clean-up phases are short and infrequent, the probability of any particular step  $t$  being in a clean-up phase is negligible. In particular, for any given step  $t$ , the backlog of the algorithm is at most  $O(\log \log n)$  with high probability in  $n$ .

## 4 Bounding the Backlog

In this section, we prove that, if the Dual-Mode Algorithm does not halt, then its backlog never exceeds  $O(\log \log n)$ , regardless of whether the filler is an adaptive adversary.

**PROPOSITION 4.1.** *Suppose  $\varepsilon \geq \frac{1}{\text{polylog}(n)}$ . If the Dual-Mode Emptying Algorithm has not yet halted by some step  $t$ , then the backlog at step  $t$  is deterministically  $O(\log \log n)$ .*

To prove Proposition 4.1, we begin by demonstrating several key properties of the algorithm. First we show that, whenever the Dual-Mode Algorithm removes water from a cup  $j$ , it is always able to remove a full unit of water.

**LEMMA 4.1.** *Whenever a cup is about to be emptied from by the Dual-Mode Algorithm, the cup contains at least 1 unit of water.*

*Proof.* Consider a round  $r$ , and let  $t_0$  be the final step prior to round  $r$ . Let  $t$  be a step in round  $r$  and  $j$  be a cup. The number of random thresholds crossed in cup  $j$  by step  $t$  is at most

$$(4.1) \quad |\{\tau_j(1), \dots, \tau_j(\lceil g_j(t_0) \rceil - 1)\}| \leq g_j(t_0).$$

If the total amount of water ever placed into cup  $j$  by step  $t$  is  $f_j$ , then the number of times that water is removed greedily from cup  $j$  during steps  $1, 2, \dots, t$  is,

$$(4.2) \quad f_j - g_j(t) \leq f_j - g_j(t_0).$$

By (4.1) and (4.2), the total number of times that cup  $j$  is emptied from during steps  $1, 2, \dots, t$  is at most  $f_j$ . If water is removed from cup  $j$  during step  $t$ , then the total number of empties from cup  $j$  prior to step  $t$  is at most  $f_j - 1$ . Thus cup  $j$  contains at least 1 unit of water prior to being emptied out of during step  $t$ .  $\square$

One consequence of Lemma 4.1 is that the number of greedy-mode steps that remove water from a cup  $j$  equals the total amount of water removed from cup  $j$  in greedy mode. Thus the crossing weight  $g_j(t)$  of a cup  $j$  after step  $t$  is equal to the total amount of water placed into cup  $j$  in steps  $1, 2, \dots, t$ , minus the total amount removed in greedy mode from cup  $j$  during steps  $1, 2, \dots, t$ . Throughout the rest of the analysis of the algorithm, we will often implicitly use this definition of crossing weight in place of the original.

The second property of the algorithm that we prove is that, at the beginning of each round, there are only  $O(1)$  units of water in each cup. In fact, we prove a stronger version of this statement which will be useful again later in the algorithm analysis.

**LEMMA 4.2.** *At the beginning of each round  $r$ , the amount of water in each cup  $j$  is less than  $2 + z$ , where  $z = g_j(t_0) - \lfloor g_j(t_0) \rfloor$  is the non-integer part of cup  $j$ 's crossing weight after the final step  $t_0$  preceding round  $r$ .*

*Proof.* Call a threshold  $\tau_j(i)$  **closed** at the end of a step  $t$  if  $g_j(t) \geq i+1$ . If a threshold  $\tau_j(i)$  is closed and cup  $j$  does not require greedy emptying, then threshold  $\tau_j(i)$  is guaranteed to have already been scheduled to some slot. Moreover, at the beginning of any round  $r$ , and for each cup  $j$ , there can be at most one closed threshold  $\tau_j(i)$  whose scheduled slot has not yet occurred. In particular, if two crossings occur in cup  $j$  during round  $r-1$ , the second crossing must be of a threshold  $\tau_j(i)$  that is *not* yet closed by the start of round  $r$ . (In particular, if the threshold were closed, cup  $j$  would have required additional greedy emptying in round  $r-1$ .)

Consider the steps  $1, 2, \dots, t_0$  prior to round  $r$ , and let  $f_j$  denote the total amount of water placed into cup  $j$  during steps  $1, 2, \dots, t_0$ . By the beginning of round  $r$ , there are  $\lfloor g_j(t_0) \rfloor - 1$  closed random thresholds  $\tau_j(i)$  in cup  $j$ . Since at most one of those thresholds can be scheduled to a slot that has not yet occurred, the total number of prescheduled-mode empties from cup  $j$  during steps  $1, 2, \dots, t_0$  is at least  $\lfloor g_j(t_0) \rfloor - 2$ . The total number of greedy-mode empties from cup  $j$  during steps  $1, 2, \dots, t_0$  is  $f_j - g_j(t_0)$ , and thus the total number of (any-mode) empties from cup  $j$  during steps  $1, 2, \dots, t_0$  is at least

$$f_j - g_j(t_0) + \lfloor g_j(t_0) \rfloor - 2 = f_j - z - 2.$$

By Lemma 4.1, it follows that cup  $j$  can contain at most  $2+z$  units of water at the end of step  $t_0$ .

□

Lemma 4.2 tells us that the only way a cup can have large fill during a round  $r$  is if it receives almost all of that fill *during* round  $r$ . Recall that whenever a cup receives a large amount of water during a round, the cup is said to require greedy emptying. To prove a bound on backlog, we therefore wish to prove that the greedy mode of the Dual-Mode Algorithm keeps the backlog of cups that require greedy emptying small. The key step in doing this is to bound the number of distinct cups that ever require greedy emptying during a round  $r$ .

**LEMMA 4.3.** *Suppose  $\ell \leq \text{polylog}(n)$  and  $\varepsilon \geq \frac{1}{\text{polylog}(n)}$ , where  $\ell$  is the number of prescheduled slots in each round (i.e., the number of steps ignoring greedy-mode steps) and  $\varepsilon$  is the resource-augmentation parameter. Then during a given round  $r$ , at most  $\text{polylog}(n)$  distinct cups ever require greedy emptying.*

*Proof.* In order for a cup  $j$  to require greedy emptying, the filler must place at least 1 unit of water into cup  $j$  during round  $r$ . To bound the number of cups that ever require greedy emptying it therefore suffices to bound the length of each round  $r$  by  $\text{polylog}(n)$ . Since the number of prescheduled steps is  $\ell \leq \text{polylog } n$ , we focus on the number of greedy-mode steps.

For each step that the algorithm is in greedy mode, the algorithm removes  $\varepsilon$  more water than the filler inserts (by Lemma 4.1). On the other hand, the filler never greedily removes more water from a cup  $j$  than was placed into that cup during the round. It follows that if the Dual-Mode Algorithm is in greedy mode for  $k$  steps of a round, then the filler must insert at least  $\varepsilon k$  units of water into cups during the non-greedy-mode steps. This implies that  $\varepsilon k \leq \ell \leq \text{polylog}(n)$ . Since  $\varepsilon \geq \frac{1}{\text{polylog}(n)}$ , it follows that  $k \leq \text{polylog}(n)$ , as desired. □

Lemmas 4.2 and 4.3 combine to imply that only a small number  $\text{polylog}(n)$  of cups can contain more than a few units of water at any given moment. By employing the standard analyses [1, 11, 14] of the greedy algorithm on these cups, we can bound the backlog and prove Proposition 4.1.

*Proof.* [Proof of Proposition 4.1] By Lemmas 4.2 and 4.3, the number of cups at any step that contain a large amount of water, say more than 5 units, can never exceed  $\text{polylog}(n)$ . Moreover, whenever such a cup exists, it is guaranteed to require greedy emptying, meaning that the emptying algorithm always removes water from the fullest such cup. If we consider only the water above height 5 in each cup, then we have a cup game in which there are never more than  $q = \text{polylog}(n)$  cups and in which the emptying algorithm always removes from the fullest cup. By Lemma 1 of Bender et al. [11], which analyzes the greedy emptying algorithm for the so-called dynamic cup game, it follows that the backlog never exceeds  $O(\log q) = O(\log \log n)$ .<sup>5</sup> □

## 5 Bounding the Probability of Halting

In this section, we bound the probability of the Dual-Mode Algorithm halting.

**PROPOSITION 5.1.** *Suppose  $\varepsilon \geq \frac{1}{\text{polylog}(n)}$ , let  $p = \frac{1}{\text{poly}(n)}$  and let  $m = \text{poly}(n)$ . For a cup game on  $n$  cups with resource-augmentation parameter  $\varepsilon$ , suppose the emptier follows the Dual-Mode Emptying Algorithm using  $\ell = \log^c n$  for  $c$  a sufficiently large constant, and suppose the filler is an elevated adaptive adversary. With probability at least  $1 - p$ , the emptying algorithm does not halt in the first  $m$  steps.*

An important idea in proving Proposition 5.1 is to reduce the elevated adaptive filler to a simpler adversary, that we call an **integer-updated filler**. The integer-updated filler does not explicitly see any cups. Rather, at certain points in time, the integer-updated filler is updated with scheduling decisions made by the Dual-Mode Algorithm.

<sup>5</sup>In place of Lemma 1 of Bender et al. [11], one can also use the older analysis of Adler et al. [1] for the greedy algorithm in the (non-dynamic) cup game. The only difference is that, rather than applying the greedy analysis to the entire game, one must separately apply it to each round of the game.

The integer-updated filler gets its name from the fact that it is given updates whenever a cup crosses an “integer threshold”. Say that a cup *crosses an integer threshold*  $k \in \mathbb{N}$  during a step  $t$ , if  $g_j(t-1) < k \leq g_j(t-1) + c_j(t)$ , where  $c_j(t)$  is the amount of water placed into the cup by the filler during step  $t$ . Note that, equivalently, an integer threshold is crossed at step  $t$  if and only if the amount of water in cup  $j$  increases to be greater than or equal to an integer  $k'$  that it was previously less than. Whenever a cup crosses an integer threshold  $k > 1$ , the Dual-Mode Algorithm learns either that (a) the cup requires greedy emptying; or (b) the round number  $r'$  and slot  $i'$  to which threshold  $\tau_j(k-1)$  is scheduled.

Lemma 5.1 shows that the integer-updated filler is at least as powerful an adversary against the Dual-Mode Algorithm as is the elevated adaptive filler.

**LEMMA 5.1.** *Consider a cup game in which the emptier follows the Dual-Mode Emptying Algorithm, and the filler is integer-updated. During and after each step  $t$ , the integer-updated filler can determine which cups contain fill greater than 3, and how much water is in each of those cups.*

*Proof.* If at a given step  $t$  and round  $r$ , a cup  $j$  has already crossed at least one integer threshold during the round, then the cup is said to be an **active cup**. By Lemma 4.2, inactive cups cannot contain more than 3 units of water. Thus it suffices to show that the filler can reconstruct the fills of the active cups.

We claim that, at each step  $t$ , the integer-updated filler can reconstruct all greedy empties performed so far by the Dual-Mode Algorithm, and can determine the exact fills of all active cups (both during the step, when the filler has placed water but the emptier has not removed water, and after the step). Suppose by induction that this holds after step  $t-1$ , and that we wish to prove it after step  $t$ .

Let  $r$  be the round containing step  $t$ . If a cup  $j$  is active after step  $t$ , then the integer-updated filler is aware of all slots in rounds  $1, 2, \dots, r$  that the cup  $j$  has ever been scheduled to. By the inductive hypothesis, the filler also knows which steps  $1, 2, \dots, t-1$  have corresponded with which slots of each round; and which steps have performed greedy empties from cup  $j$ . Thus the filler can deduce the full history of cup  $j$  in steps  $1, 2, \dots, t-1$ , and can also determine whether cup  $j$  requires greedy emptying during step  $t$ . If at least one active cup requires greedy emptying during step  $t$ , then the filler can determine which is fullest, and thus which cup the emptier empties out of; note that any cup that requires greedy emptying is guaranteed to be active. On the other hand, if no active cups require greedy emptying, then the filler can determine whether any active cups are scheduled to be emptied out of during the step (since the filler is aware of all scheduling decisions that have been made for active cups). Thus the filler can reconstruct all greedy empties

performed in steps  $1, 2, \dots, t$ , and the exact fills of all greedy cups during and after step  $t$ .  $\square$

We analyze the Dual-Mode Emptying Algorithm against an integer-updated adversary by comparing the Dual-Mode Algorithm’s behavior to that of a second algorithm, which we call the Delayed-Scheduling Dual-Mode Algorithm (or the Delayed-Scheduling Algorithm for short). The Delayed-Scheduling Algorithm has certain “super powers” that allow for it to schedule events *after* those events have occurred. We prove that the integer-filled adversary is essentially unable to distinguish between the Dual-Mode and the Delayed-Scheduling Algorithms; this allows us to then complete the analysis of the Dual-Mode Algorithm by analyzing the halting probability of the Delayed-Scheduling Algorithm.

**The Delayed-Scheduling Algorithm.** Whereas the Dual-Mode Algorithm performs a scheduling whenever a threshold-crossing occurs, the Delayed-Scheduling Algorithm delays making a scheduling decision until it needs to reveal that decision to the integer-updated filler. That is, whenever the crossing weight of a cup  $j$  crosses an integer threshold  $k > 1$ , and the cup does not require greedy emptying, the algorithm retroactively decides on the value of the random threshold  $\tau_j(k-1)$ ; from this the algorithm can determine which round  $r$  the threshold  $\tau_j(k-1)$  should be scheduled to (note that  $r$  may be a round that’s already occurred!), and the algorithm assigns  $\tau_j(k-1)$  to a random slot in round  $r$ , out of those not yet scheduled. The pseudocode for each round of the Delayed-Scheduling Dual-Mode Algorithm is given by Algorithm 2.

A peculiar property of the Delayed-Scheduling Dual-Mode Algorithm is that it sometimes schedules thresholds to slots that have *already occurred*. If the filler were an elevated adaptive adversary, then the fact that the Delayed-Scheduling Algorithm can “overwrite the past” would introduce circular dependencies that prevent the algorithm from being well defined. In particular, the filler might be able to adapt to scheduling decisions that the emptier has not actually made yet. On the other hand, if the filler is integer-updated, then the only information available to the filler at a given point in time is precisely the set of scheduling decisions that the emptier has already finalized. This ensures that there cannot be circular dependencies and that the Delayed-Scheduling Algorithm is well defined. Although the Delayed-Scheduling Algorithm is impossible to implement (because it retroactively changes the past), it serves as a useful tool for analyzing the Dual-Mode algorithm.

The random procedure used by the Dual-Mode Algorithm to assign random thresholds to slots (i.e., selecting a random slot out of the remaining unscheduled ones) is designed to make the Dual-Mode and Delayed-Scheduling Dual-Mode Algorithms difficult to distinguish between for an integer-updated filler. In particular, if the filler has been

---

**Algorithm 2** A Round of the Delayed-Scheduling Dual-Mode Emptying Algorithm

---

**Description:** The round begins with step  $t_0 + 1$ . The random thresholds  $\tau_j(i)$  are determined on the fly by the algorithm (unlike for the Dual-Mode Emptying Algorithm). We use  $\mathcal{A}_s$  to denote the set of scheduling assignments for a given round  $s$ . (i.e.,  $(i, j) \in \mathcal{A}_s$  if the  $i$ -th slot of round  $s$  empties from cup  $j$ .)

The set of assignments  $\mathcal{A}_s$  for a given round  $s$  may be partially determined *after* the completion of round  $s$ . In this case, the algorithm retroactively changes its behavior in the past.

```

1:  $i \leftarrow 1 \triangleright i - 1$  is the number of prescheduled-mode steps so far.
2:  $t \leftarrow t_0 \quad \triangleright t$  is the current step number.
3:  $b_j \leftarrow g_j(t_0)$  for each cup  $j \triangleright b_j$  is the crossing weight of cup  $j$  prior to the round beginning.
4: while  $i \leq \ell$  do
5:    $t \leftarrow t + 1$ 
6:   Filler takes a turn placing some amount  $c_j(t)$  of water into each cup  $j$ 
7:   for  $j$  s.t.  $b_j \geq 1$  and  $g_j(t - 1) < \lceil b_j \rceil \leq g_j(t - 1) + c_j(t)$  do
8:      $\tau_j(\lceil b_j \rceil - 1) \leftarrow$  random real number in  $[\lceil b_j \rceil - 1, \lceil b_j \rceil)$ 
9:      $s \leftarrow$  the round in which cup  $j$  crossed threshold  $\tau_j(\lceil b_j \rceil - 1)$ 
10:    if  $|\mathcal{A}_{s+1}| \geq \ell$  then Halt end if
11:    Add  $(a, j)$  to  $\mathcal{A}_{s+1}$  for a random  $a \in [\ell] \setminus \{a \mid (a, d) \in \mathcal{A}_{s+1} \text{ for some } d\}$ .
12:    if  $a$ -th slot of round  $s + 1$  has already happened then
13:      Travel back in time and remove 1 unit of water from cup  $j$  during that slot.
14:    end if
15:  end for
16:  if  $g_j(t - 1) + c_j(t) \geq \lceil b_j \rceil + 1$  for some cup  $j$  then
17:     $\triangleright$  Operate in greedy mode.
18:    Let  $j$  be the fullest such cup and remove 1 unit of water from cup  $j$ 
19:  else
20:     $\triangleright$  Operate in prescheduled mode
21:    if  $(i, j) \in \mathcal{A}_r$  for some  $j$  then
22:      Remove 1 unit of water from cup  $j$ .
23:    end if
24:     $i \leftarrow i + 1$ 
25:  end if
26: end while

```

---

informed about the schedulings of some set of  $k$  random thresholds to slots in a round  $r$ , then the filler cannot distinguish between whether those are the only  $k$  slots that have been scheduled in round  $r$  (as is the case for the Delayed-Scheduling Algorithm); or whether the filler is simply seeing a subset of the slots that have been scheduled to round  $r$  (as is the case for the original algorithm). In either case, the  $k$  slots that have been scheduled to are simply  $k$  random slots out of the  $\ell$  total slots for round  $r$ .

Note that there is an important way in which the Delayed-Scheduling Dual-Mode Algorithm is not simply a reframing of the Dual-Mode Algorithm, even against an integer-updated filler. In particular, whenever a round  $r$  has more than  $\ell$  thresholds scheduled to it, the Dual-Mode Algorithm halts in round  $r - 1$ . In contrast, in the same situation, the Delayed-Scheduling Algorithm may continue past round  $r$ , and during round  $r$  the algorithm “sees into the future”, by determining which are the first  $\ell$  thresholds that the integer-updated filler will learn are scheduled to round  $r$ , and allocating the slots in round  $r$  to those thresholds. This makes it so that the Delayed-Scheduling Algorithm only halts when the integer-updated filler is about to learn about the  $\ell + 1$ -th threshold scheduled to round  $r$  (which may be well after round  $r$ ).

Lemma 5.2 uses a coupling argument to compare the Dual-Mode Algorithm and the Delayed-Scheduling Dual-Mode Algorithm against an integer-updated filler adversary. We prove that, in order to bound the probability of the Dual-Mode Algorithm halting, it suffices to instead analyze the more powerful (and impossible to actually implement) Delayed-Scheduling Dual-Mode Algorithm.

**LEMMA 5.2.** *Let  $m$  be a game length. Let  $\mathcal{A}$  be a (possibly randomized) integer-updated filling algorithm, and suppose that  $\mathcal{A}$  has probability  $p$  of causing the Dual-Mode Algorithm to halt in the first  $m$  steps.*

*Then there is an integer-updated filling algorithm algorithm  $\mathcal{B}$  that, has probability at least  $p$  of causing the Delayed-Scheduling Dual-Mode Emptying Algorithm to halt in the first  $m + n$  steps.*

*Proof.* We define the filling algorithm  $\mathcal{B}$  to first follow algorithm  $\mathcal{A}$  for the first  $m$  steps, and to then spend the final  $n$  steps adding additional water to each cup  $j$  so that  $g_j(n + m)$  is an integer. (By Lemma 4.1, this is equivalent to the total amount of water ever placed in cup  $j$  being an integer.)

We prove the lemma by constructing a coupling of two multi-step random processes. The first process  $X$  is the process of algorithm  $\mathcal{A}$  running against the Dual-Mode Emptying Algorithm; the second process  $Y$  is the process of filling algorithm  $\mathcal{B}$  running against the Delayed-Scheduling Dual-Mode Emptying Algorithm. Consider a coupling of  $X$  and  $Y$  in which the two processes use the

same random bits to determine all random thresholds  $\tau_j(i)$ ; and in which process  $Y$  “mimics” process  $X$ ’s scheduling decisions. Specifically, at each step  $t$ , if process  $X$  has not halted or terminated yet, and if process  $Y$  needs to schedule threshold  $\tau_j(i)$  to some slot of a round  $r$ , process  $Y$  simply uses whatever slot was already assigned by process  $X$ ; if, on the other hand, process  $X$  has halted or terminated, then process  $Y$  performs its slot assignments at random (out of the remaining unscheduled slots). Additionally, if algorithm  $\mathcal{A}$  is randomized, then algorithm  $\mathcal{B}$  uses the same random bits as  $\mathcal{A}$  in the coupling.<sup>6</sup>

The key to proving that this is a valid coupling (i.e., that each time  $Y$  mimics one of  $X$ ’s decisions,  $Y$  is selecting a slot at random from the slots not yet scheduled to) is the following simple observation: If  $x_1, x_2, \dots, x_k$  are a sequence of random variables in  $[\ell]$  such that each  $x_i$  is selected at random from  $[\ell] \setminus \{x_1, \dots, x_{i-1}\}$ , then for any fixed permutation  $\pi \in S_k$ , the sequence  $x_{\pi(1)}, x_{\pi(2)}, \dots, x_{\pi(k)}$  is also a sequence of random variables with the property that each  $x_{\pi(i)}$  is uniformly random in the set  $[\ell] \setminus \{x_{\pi(1)}, \dots, x_{\pi(i-1)}\}$ . Call this the **Permutation Observation**.

Suppose process  $X$  runs for  $t$  steps before it halts or terminates. Then, during those  $t$  steps, the set of scheduling decisions (i.e., decisions for scheduling some random threshold  $\tau_j(k)$  to some slot  $i$  of some round  $r$ ) are a subset of the set of the scheduling decisions made by  $X$ . Now consider the scheduling decisions made by  $Y$  in the first  $t$  steps. The coupling dictates that each of these scheduling decisions are made to mimic the same scheduling decision that was made in  $X$ . For each round  $r$ ,  $Y$  may schedule threshold slots to round  $r$  in a different order than does  $X$ . Nonetheless, by the Permutation Observation, each slot scheduled to by  $Y$  is selected uniformly at random from the not-yet-scheduled slots in round  $r$ . Therefore the emptying algorithm performed in  $Y$  is a valid instance of the Delayed-Scheduling Emptying Algorithm, meaning that the proposed coupling is well defined.

Using the coupling, we next show that if process  $X$  halts in the first  $m$  steps, then process  $Y$  halts in the first  $n + m$  steps. Suppose process  $X$  halts in the first  $m$  steps. Then there is some round  $r$  to which  $X$  is forced to schedule more than  $\ell$  thresholds. Since  $X$  and  $Y$  use the same random thresholds and since  $Y$  finishes step  $m + n$  with  $g_j(m + n)$  an integer for each cup  $j$ , every threshold that  $X$  schedules to some round  $r$  is also scheduled by  $Y$  to round  $r$  by the end of step  $m + n$ . (In particular, because  $g_j(m + n)$  is an integer,  $Y$  cannot still be delaying the scheduling of the threshold at step  $m + n$ .) Thus process  $Y$  schedules more than  $\ell$  thresholds to

round  $r$ , meaning that  $Y$  also halts.

We have shown that the probability of  $Y$  halting in the first  $m + n$  steps is at least as large as the probability of  $X$  halting in the first  $m$  steps, completing the proof.  $\square$

Lemma 5.3 proves that the Delayed-Scheduling Algorithm is highly resistant to an integer-updated filler trying to cause the algorithm to halt. The basic idea is, that by the time the integer-updated filler learns about a random threshold crossing  $\tau_j(i)$ , it is too late for that information to be useful. Because the integer-updated filler is unable to infer useful information about the random thresholds  $\tau_j(i)$ , they are unable to reliably cause a large number of random threshold crossings to occur in a given round.

**LEMMA 5.3.** *Let  $m \leq \text{poly}(n)$ ,  $1/\text{polylog}(n) \leq \varepsilon \leq 1/2$ , and  $\ell = \log^c n$  for a sufficiently large constant  $c$  (as a function of  $m$  and  $\varepsilon$ ). Then any integer-updated filling algorithm has probability at most  $1/n^c$  of causing the Delayed-Scheduling Dual-Mode Emptying Algorithm to halt during an  $m$ -step game.*

*Proof.* For a real number  $x$ , let  $\bar{x} = \lceil x \rceil - x$  denote the amount that must be added to  $x$  to reach the next integer, and let  $\underline{x} = x - \lfloor x \rfloor$  denote the amount that must be subtracted from  $x$  to reach the largest integer less than or equal to  $x$ .

Consider a round  $r$  beginning after some step  $t_0$  and ending at some step  $t_1$ , and let us bound the probability that the filler is able to cross more than  $\ell$  random thresholds during round  $r$ . For each cup  $j$ , let  $x_j = \min(g_j(t_1) - g_j(t_0), \bar{g}_j(t_0))$  be the amount of water placed in cup  $j$  during round  $r$  that risks crossing threshold  $\tau_j(\lceil g_j(t_0) \rceil - 1)$  (although technically when  $\lceil g_j(t_0) \rceil - 1 = 0$  there is no such threshold), and let

$$y_j = \begin{cases} 0 & \text{if } g_j(t_1) \leq \lceil g_j(t_0) \rceil \\ g_j(t_1) & \text{otherwise,} \end{cases}$$

be the amount of water placed in cup  $j$  during round  $r$  that risks crossing threshold  $\tau_j(\lceil g_j(t_0) \rceil)$ .

The key feature of the Delayed-Scheduling Dual-Mode Algorithm is that  $x_j$  is fully determined by the filler before the emptying algorithm determines  $\tau_j(\lceil g_j(t_0) \rceil - 1)$  (which happens either after round  $r$  or once  $g_j(t)$  permanently increases to  $\lceil g_j(t_0) \rceil + 1$  or above); and similarly  $y_j$  is fully determined by the filler before the emptying algorithm determines  $\tau_j(\lceil g_j(t_0) \rceil)$  (which happens after round  $r$ ).

Note that  $\sum_j g_j(t_1) - \sum_j g_j(t_0) \leq (1 - \varepsilon)\ell$ . Thus the filler is forced to choose values of  $x_j, y_j \in (0, 1)$  for each  $j \in [n]$  such that

$$(5.3) \quad \sum_j (x_j + y_j) \leq (1 - \varepsilon)\ell.$$

Once a given  $x_j$  (resp.  $y_j$ ) is chosen, there is an  $x_j$  (resp.  $y_j$ ) probability of the corresponding random threshold being

<sup>6</sup>Although we consider the case where  $\mathcal{A}$  is a randomized algorithm, one can show with the Method of Conditional Probabilities that for any randomized algorithm  $\mathcal{A}$ , there is a deterministic algorithm  $\mathcal{A}'$  that has at least as high a probability of causing the Dual-Mode Emptying Algorithm to stall. Thus w.l.o.g.,  $\mathcal{A}$  is deterministic.

crossed (or 0 if there is no corresponding random threshold). Moreover, the outcome of the threshold-crossing event is determined *after* the value for  $x_j$  (resp.  $y_j$ ) is finalized by the filler.

In order to bound the probability that the filler achieves more than  $\ell$  threshold crossings during round  $r$ , we consider a relaxed version of the problem. In the relaxed problem, we allow the filler to select values for the  $x_j$ s and  $y_j$ s (in any order) subject only to the constraint (5.3) (even though in original problem there are additional constraints both on the order in which the  $x_i$ s and  $y_j$ s are finalized and on their values), and we assume that whenever the filler selects the value for an  $x_j$  or  $y_j$ , the filler is then immediately informed of whether the choice resulted in a random threshold crossing (even though in original problem, this may not be decided by the Delayed-Scheduling Algorithm until later). Claim 1 shows that, even in the relaxed problem, it is impossible for the filler to achieve more than  $\ell$  threshold crossings with any substantial probability.

**CLAIM 1.** *Let  $\mu \in [0, n]$ , and suppose that Alice is allowed to select a sequence of numbers  $p_1, p_2, \dots, p_k$ ,  $p_i \in [0, 1]$ , such that  $\sum_i p_i \leq \mu$ . Each time Alice selects a number  $p_i$ , she wins 1 dollar with probability  $p_i$ . Alice is an adaptive adversary in that she can take into account the results of the first  $i$  bets when deciding on  $p_{i+1}$ . If  $X$  is Alice's profit from the game,*

$$\Pr[X > (1 + \delta)\mu] \leq \exp(-\delta^2 \mu / 3),$$

for all  $\delta \in (0, 1)$ .

The proof of Claim 1 is deferred to Appendix A. The fact that Alice is an adaptive adversary prevents Claim 1 from being proven directly by a Chernoff bound. A natural alternative approach to proving the claim is to apply McDiarmid's inequality [31] (i.e., Azuma's Inequality applied to Doob Martingales); this also fails, however, due to the fact that McDiarmid's inequality only gives strong bounds when analyzing betting games in which the number of bets is small. Nonetheless, the moment-generating-function-based arguments on which both Chernoff bounds and McDiarmid's inequality are built can easily be adapted to prove the claim.

Applying Claim 1 to the filler's decision of values for  $\{x_j, y_j \mid j \in [n]\}$ , we can set  $X$  to be the number of threshold crossings in round  $r$ , use  $k = 2n$ , use  $\mu = (1 - \varepsilon)\ell$ , and set  $\delta = \varepsilon$  to deduce that

$$\Pr[X > \ell] \leq \exp(-\varepsilon^2(1 - \varepsilon)\ell/3) \leq \exp(-\varepsilon^2\ell/6).$$

Assuming  $\ell = \log^c n \geq (\frac{6}{\varepsilon^2})^2$ , it follows that,

$$\Pr[X > \ell] \leq \exp(-\log^{c/2} n).$$

Applying a union bound across the  $\leq m = \text{poly}(n)$  rounds of the game, the probability that any round incurs

more than  $\ell$  threshold crossings (and thus that a round halts) is at most  $m \exp(\log^{c/2} n)$ . For  $c$  a sufficiently large constant, this probability is at most  $\frac{1}{n^c}$ .  $\square$

We can now complete the proof of Proposition 5.1.

*Proof.* [Proof of Proposition 5.1] By Lemma 5.3, the proposition holds if the emptier follows the Delayed-Scheduling Dual-Mode Emptying Algorithm (instead of the Dual-Mode Emptying Algorithm). By Lemma 5.2 the proposition also holds if the emptier follows the Dual-Mode Emptying Algorithm, as desired.  $\square$

## 6 Proof of Theorem 1.1

We now combine the results from previous sections to complete the proof of Theorem 1.1. By directly applying Propositions 4.1 and 5.1, one can obtain high-probability bounds on backlog for the first  $\text{poly } n$  steps of any game. In order to give probabilistic guarantees on backlog for games of arbitrary length, we perform the Dual-Mode Algorithm repeatedly, with “clean-up” phases in-between repetitions.

*Proof.* [Proof of Theorem 1.1] Let  $\mathcal{A}$  be the Dual-Mode Algorithm with  $\ell \leq \text{polylog } n$  selected to be sufficiently large. Let  $x$  be the smallest multiple of  $\ell$  greater than  $n$ . By Proposition 5.1, we can obtain what we call the **empty-start guarantee**: if all  $n$  cups are initially empty, then the probability that  $\mathcal{A}$  halts in the first  $x + \frac{20n}{\varepsilon p}$  steps against an elevated adaptive filler is at most  $p/2$ .

By taking advantage of the structure of the Dual-Mode Algorithm  $\mathcal{A}$ , we can extend this to what we call the **non-empty-start guarantee**: if each cup  $j$  initially contains  $q_j \leq 1$  units of water, then the probability that  $\mathcal{A}$  halts in the first  $\frac{20n}{\varepsilon p}$  steps against an elevated adaptive filler is at most  $p/2$ . To prove this guarantee, imagine that the cups are initially empty and that the filler spends the first  $x$  steps placing  $q_j$  units of water in each cup  $j$ . Since the first unit of water placed in each cup does not cross any random thresholds for the Dual-Mode Algorithm, and also does not cause the cup to require greedy emptying, the Dual-Mode Algorithm  $\mathcal{A}$  will remain completely inactive during the first  $x$  steps (i.e., it will perform no greedy empties and no schedulings of thresholds), and start a new round on step  $x + 1$ . Applying the empty-start guarantee to the first  $x + \frac{20n}{\varepsilon p}$  steps, the probability of the Dual-Mode Algorithm halting in the final  $\frac{20n}{\varepsilon p}$  steps is at most  $p/2$ , which is precisely the non-empty-start guarantee.

We construct an emptying algorithm  $\mathcal{B}$  that gives probabilistic bounds on backlog for arbitrarily long games as follows. For all  $i \in \mathbb{N}$ , select a random **stopping time**  $s_i \in \{(i - 1)\frac{10n}{\varepsilon p} + 1, \dots, i\frac{10n}{\varepsilon p}\}$ . Let  $\mathcal{A}'$  be the emptying algorithm that first performs  $\mathcal{A}$  and then uses the greedy algorithm once  $\mathcal{A}$  halts. Then algorithm  $\mathcal{B}$  works as follows. Algorithm  $\mathcal{B}$  begins by running  $\mathcal{A}'$  until step  $s_1 - 1$ . Then,

at each stopping time  $s_i$ , algorithm  $\mathcal{B}$  performs the greedy emptying algorithm until reaching a step at the beginning of which every cup contains less than 1 unit of water, after which  $\mathcal{B}$  runs (a new copy of) algorithm  $\mathcal{A}'$  until step  $s_{i+1} - 1$ .

Algorithm  $\mathcal{B}$  has the property that, whenever there is a step during which some cup contains a sufficiently large constant amount of water (say, at least 5 units), the algorithm  $\mathcal{B}$  will remove a full unit of water from some cup during that step. (In particular  $\mathcal{B}$  will either be running a greedy algorithm or running  $\mathcal{A}'$  in greedy mode.) The total amount of water in all cups therefore never exceeds  $5n$ . Thus, if  $\mathcal{B}$  begins using the greedy emptying algorithm at some step  $s_i$ , then by step  $s_i + \frac{5n}{\varepsilon}$ , there will be some step after which every cup contains less than 1 unit of water (since, until such a step occurs,  $\mathcal{B}$  removes  $\varepsilon$  more water on each step than the filler puts in). Call the sequence of steps  $s_i, \dots, s_i + \frac{5n}{\varepsilon}$  the  *$i$ -th greedy window*; the sequence of steps between the  $i$ -th and  $i+1$ -th greedy windows are called the  *$i$ -th  $\mathcal{A}'$ -window*.

By Proposition 4.1, there are two ways that a step  $t$  may have backlog greater than  $O(\log \log n)$ :

- **Step  $t$  is in a greedy window:** The probability of this happening is equal to the probability that one of  $t, t-1, \dots, t - \frac{5n}{\varepsilon}$  are picked as a stopping time  $s_i$ . Note that if  $t$  is in the *final* step of a greedy window, then the backlog is guaranteed to be  $O(1)$ , so to avoid a fence-post error, we will bound only the probability that one of  $t, t-1, \dots, t - \frac{5n}{\varepsilon} + 1$  is picked as a stopping time  $s_i$ . Since each step  $k$  has probability  $\frac{\varepsilon p}{10n}$  of being picked as a stopping time  $s_i$ , the probability of any of  $t, t-1, \dots, t - \frac{5n}{\varepsilon} + 1$  being picked is at most  $p/2$ .
- **Step  $t$  is in an  $\mathcal{A}'$ -window in which  $\mathcal{A}'$  halts.** By the non-empty starting guarantee, since each  $\mathcal{A}'$ -window has length at most  $\frac{20n}{\varepsilon p}$ , the probability of  $\mathcal{A}'$  halting in any given  $\mathcal{A}'$ -window is at most  $p/2$ .

Combining the two cases, the probability that step  $t$  has backlog greater than  $O(\log \log n)$  is at most  $p$ , as desired.  $\square$

## 7 An Application to Data Structures

The analysis of the deterministic greedy emptying algorithm [1, 14], which appeared in its earliest form nearly thirty years ago, has found extensive applications in data-structure deamortization [3, 4, 14–16, 21, 24, 33]. Until recently, the main obstacle to applying *randomized* emptying algorithms in similar ways has been that the only known randomized algorithms required substantial resource augmentation [15] (in fact, the emptier needed to be able to fully empty a cup at each step). This changed with the recent work of Bender, Farach-Colton, and Kuszmaul [11], which gives randomized emptying algorithms requiring resource augmentation only

$$\varepsilon = \frac{1}{\text{polylog } n}.$$

**A second obstacle: cyclic dependencies.** Applying randomized cup emptying directly to data-structure design remains highly nontrivial, however, due to the requirement that the cup-game filler be an oblivious adversary. If one models part of a data structure  $D$  as a cup game, and uses a randomized cup emptying strategy within that cup game, then the internal workings of the data structure  $D$  may allow for the randomized emptying decisions in the cup game to indirectly later affect the input to the same cup game. This is known as a *cyclic dependency* [10], and means that the “filler” for the cup game is not oblivious. Because the filler is not oblivious, the guarantees for the randomized cup emptying algorithm may no longer hold. In other words, cyclic dependencies make the use of standard randomized cup emptying algorithms impossible.

A recent application of randomized cup games to data structures [10] navigated this obstacle by substantially redesigning the data structure in order to eliminate cyclic dependencies. Here, we suggest an alternative framework for resolving cyclic dependencies using the Dual-Mode Algorithm.

**Collapsing cyclic dependencies with an integer-updated filler.** Recall that the Dual-Mode Algorithm (as well as the long-running variant given in Theorem 1.1) achieves backlog guarantees against an integer-updated filler. This means that, for the Dual-Mode Algorithm, certain cyclic dependencies are acceptable. In particular, the empties performed on a cup  $j$  prior to some time  $t$  are allowed to subsequently effect the input to the system, as long as cup  $j$  has either crossed an integer threshold since time  $t$ , or has required greedy emptying at some point since step  $t$ .<sup>7</sup> This suggests a natural approach to resolving cyclic dependencies: whenever the previous behavior of a cup  $j$  is about to effect the input to the cup game, simply delay that effect until the next time cup  $j$  crosses an integer threshold or enters greedy mode. Although this delay could be for a substantial amount of time, a key property is that cup  $j$  will be emptied out of at most twice during the delay. We call this the *small-height delay property*.

**7.1 Low latency  $B^\varepsilon$ -trees** In this section, we summarize a recent application of randomized cup emptying to  $B^\varepsilon$ -tree flushing strategies [10], and describe how the Dual-Mode Algorithm can be used in this application to simplify the handling of cyclic dependencies.

**$B^\varepsilon$ -trees.** The  $B^\varepsilon$ -tree is an external memory dictionary

<sup>7</sup>Note that, in order for cup  $j$  to require greedy emptying at some point since time  $t$ , but to have not yet crossed an integer threshold, cup  $j$  must have required greedy emptying *at* time  $t$ , meaning that the filler was already fully aware of all empties performed in cup  $j$  prior to time  $t$ .

that supports the same operation set as the standard B-tree, while offering faster amortized insertion/deletion/update costs [10]. Rather than inserting new records in the leaves of the tree (as is done for B-trees), new records are inserted into a buffer in the *root* of the tree. Whenever a buffer of any node overflows (i.e., contains  $c \cdot B$  records, where  $B$  is the block size and  $c$  is a large constant), some subset of the records in the buffer are **flushed** to the buffers of child nodes. The strategy for selecting which records to flush is called the **flushing strategy**. The goal of a flushing strategy is to flush as many records as possible to as few children as possible, thereby amortizing the I/Os (required to load and modify the children nodes) to the large number of records being flushed. Depending on the operation workload, different flushing strategies may achieve better performance than others [10, 12].

In order to allow  $B^\varepsilon$ -trees to obtain an asymptotic speedup over B-trees, one additional modification is needed. Rather than allowing nodes to have degree  $\Theta(B)$ , as in the standard B-tree, one restricts the maximum degree of each node to  $\Theta(B^\varepsilon)$  for some constant  $0 < \varepsilon < 1$ . This means that, whenever a node's buffer overflows, the buffer contains many more records than there are children to be flushed to, enabling many records to be flushed simultaneously. Moreover, since the height of the tree is preserved up to a constant factor, lookup operations still take time  $O(\log_B n)$ . Insertion operations, on the other hand, require only amortized time  $O(\frac{\log_B n}{B^{1-\varepsilon}})$ . In the same way that insertions are sped up, updates and deletions can also be made to run in amortized time  $O(\frac{\log_B n}{B^{1-\varepsilon}})$ .

**A problem: flushing cascades.** When a node  $x$  overflows, the flushing strategy may wish to aggressively flush many records to a small number of children. If it does this, however, then the buffers of those children may overflow and also require flushing. These children may, in turn, cause many grandchildren nodes to overflow, etc. This is called a **flushing cascade**.

For many natural flushing strategies, flushing cascades can make the worst-case running time of an operation be polynomial [10]. On the other hand, the standard flushing strategies that avoid flushing cascades are non-aggressive in their flushing, which can hurt the amortized performance of the data structure on natural workloads [10, 12].

**Achieving low latency for any flushing strategy.** Recently, Bender et al. [10] showed how to adapt the  $B^\varepsilon$ -tree in order to eliminate flushing cascades for *arbitrary* flushing strategies.<sup>8</sup>

<sup>8</sup>Bender et al. [10] introduce a single “universal flushing strategy” that can be parameterized to simulate any other flushing strategy. The universal flushing strategy has the attractive property that it always selects exactly  $B$  elements out of the  $O(B)$ -element buffer to flush. We implicitly assume this property in our discussion of [10]’s techniques.

They treat the nodes in each layer of the tree as a cup game, where 1 unit of water in a cup corresponds with  $O(B)$  elements in some node’s buffer. Whenever a node is flushed out of at level  $\ell$ , that corresponds with a unit of water being emptied from some cup in the cup game at level  $\ell$  (and being poured into the next level). The tree is implemented so that, whenever a flush is performed at the top level, exactly one flush is then performed at each of the lower levels.

In order to successfully apply this idea to  $B^\varepsilon$ -trees, two technical insights are needed [10]:

- **Resource augmentation can be simulated:** By defining “one unit” of water at each level  $\ell$  to be a factor of  $1 + \frac{1}{\text{polylog } n}$  larger than “one unit” of water at level  $\ell - 1$ , one can make it so that each flush from level  $\ell$  only inserts  $\leq 1 - \frac{1}{\text{polylog } n}$  water into level  $\ell + 1$ . This allows for cup emptying algorithms that rely on  $\varepsilon = \frac{1}{\text{polylog } n}$  resource augmentation to be used in each cup game.
- **Large buffers can be stored in main memory:** A key property of the cup emptying algorithm used by Bender et al. [10] is that, in addition to cups having backlog bounded by  $O(\log \log n)$  (with high probability), the *total* amount of water in cups of height greater than  $\Omega(1)$  is  $\text{polylog } n$ . As a consequence, even though the cup-flushing-based  $B^\varepsilon$ -tree may allow for a node’s buffer to overflow past  $O(B)$  elements (i.e., the corresponding cup has more than  $O(1)$  units of water), the excess overflow for *all* nodes can be stored in a small in-memory cache, preventing the overflows from hurting performance. Without this cache, the time to access an overflowed node would be superconstant, destroying the theoretical guarantees of the data structure.

Note that the Dual-Mode Algorithm (and its long-running counterpart given in Theorem 1.1) also has this property. In particular, since each cup has fill at most 3 at the beginning of each round, and each round is at most  $\text{polylog } n$  steps, the number of cups containing fill greater than 4 will never exceed  $\text{polylog } n$  (at least, with high probability at each step), and the amount of water in each such cup will not exceed  $O(\log \log n)$  (also with high probability).

**Handling structural changes to the tree.** The internal structure of a  $B^\varepsilon$ -tree may change substantially over time. In particular, whenever a node  $x$  has too many children, that node  $x$  splits into two nodes  $x_1$  and  $x_2$ . Similarly, if the number of children for a node  $x$  becomes too small, that node  $x$  may either borrow children from a neighbor or merge with a neighbor node.

For brevity, in the rest of our discussion here, we will consider only insertions and not deletions. Deletions can also be handled with similar techniques.

Considering only insertions, the only way that a node  $x$  can be modified is if it's forced to split into two nodes  $x_1$  and  $x_2$ . When this happens, two new empty buffers (i.e., empty cups) are created for  $x_1$  and  $x_2$ , and node  $x$ 's buffer is *retired* (meaning that it can still be emptied from but no more water is poured into it). Whenever flushes are performed from  $x_1$  or  $x_2$ , a small number of elements are also “trickled” down the tree from  $x$ 's buffer [10]. This guarantees that by the time  $x_1$  or  $x_2$  split,  $x$ 's buffer will have been fully eliminated. Thus the total number of buffers (i.e., cups) never exceeds  $O(n)$ .

**Structural changes cause cyclic dependencies.** The emptying decisions in some cup  $j$  corresponding with some node  $x$  at level  $\ell$  of the tree, affect the structure of the tree rooted at  $x$ . This, in turn, affects when node  $x$  is next forced to split into two nodes. When  $x$  splits into two nodes, the cup  $j$  also splits into two cups, and this affects the behavior of the “filler” for the cup game at level  $\ell$ . The fact that node  $x$  has split in two may also later cause higher nodes in the tree to split, also affecting the behavior of “filler” at level  $\ell$ .

**Eliminating cyclic dependencies.** Bender et al. [10] give an elegant solution to this problem by substantially redesigning the balancing-mechanism in the  $B^\varepsilon$ -tree to eliminate cyclic dependencies. (This is also what allows for them to also handle cyclic dependencies for deletions without requiring tree rebuilds.)

Our results suggest a simple alternative solution to the cyclic dependencies. By using the version of Dual-Mode algorithm given in Theorem 1.1, we make it so that the “filler” for each cup game in the tree needs not be oblivious; instead it can be integer-updated.

Rather than eliminating the cyclic dependencies, we modify them so that the Dual-Mode Emptying Algorithm can ignore them. Whenever a node  $x$  has too many children and the node wishes to split, we simply *delay* that split until the node  $x$  either crosses its next integer threshold or enters greedy mode. By the small-height delay property, at most  $O(B)$  records will be flushed out of node  $x$  during this delay, which does not substantially affect the size of  $x$ 's subtree; this ensures that the tree remains balanced despite the delayed node splits. The delays ensure that emptying decisions at a node  $x$  at level  $\ell$  cannot affect the input to the cup game at level  $\ell$  until after the node  $x$  has crossed its next integer threshold or entered greedy mode. This means that the filler for the cup game at each level  $\ell$  is no stronger than an integer-updated filler.

In order for the Dual-Mode Emptying Algorithm to be used at each level of the tree, the emptier must store the state of the algorithm at each step (i.e., which slots are scheduled to which cups). This can be done using polylogarithmic in-memory space. Because the filler for the cup game at each level of the tree is no stronger than an integer-updated

filler, the randomized bounds for the Dual-Mode Emptying Algorithm hold without modification, eliminating the cyclic-dependencies problem.

## Acknowledgments

The authors would like to thank Paul T. Crews for several useful conversations about applications of adaptivity to real-world systems.

This research was supported in part by NSF grants CCF-1725543, CSR-1763680, CCF-1716252, CCF-1617618, CNS-1938709, XPS-1533644. The research was also supported in part by the United States Air Force Research Laboratory and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

The author William Kuszmaul was supported in part by an NSF Graduate Fellowship and a Hertz Foundation Fellowship.

## References

- [1] M. Adler, P. Berenbrink, T. Friedetzky, L. A. Goldberg, P. Goldberg, and M. Paterson. A proportionate fair scheduling rule with good worst-case performance. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 101–108, 2003.
- [2] M. I. Alipio and N. M. C. Tiglao. Towards a taxonomy of cache-based transport protocols in wireless sensor networks. In *2017 IEEE Sensors Applications Symposium (SAS)*, pages 1–6. IEEE, 2017.
- [3] A. Amir, M. Farach, R. M. Idury, J. A. L. Poutré, and A. A. Schäffer. Improved dynamic dictionary matching. *Inf. Comput.*, 119(2):258–282, 1995.
- [4] A. Amir, G. Franceschini, R. Grossi, T. Kopelowitz, M. Lewenstein, and N. Lewenstein. Managing unbounded-length keys in comparison-driven data structures with applications to online indexing. *SIAM Journal on Computing*, 43(4):1396–1416, 2014.
- [5] Y. Azar and A. Litichevsky. Maximizing throughput in multi-queue switches. *Algorithmica*, 45(1):69–90, 2006.
- [6] A. Bar-Noy, A. Freund, S. Landa, and J. S. Naor. Competitive on-line switching policies. In *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 525–534, 2002.
- [7] A. Bar-Noy, A. Nisgav, and B. Patt-Shamir. Nearly optimal perfectly periodic schedules. *Distributed Computing*, 15(4):207–220, 2002.
- [8] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, Jun 1996.

- [9] S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. Fast scheduling of periodic tasks on multiple resources. In *Proceedings of the 9th International Parallel Processing Symposium*, pages 280–288, 1995.
- [10] M. Bender, R. Das, M. Farach-Colton, R. Johnson, and W. Kuszmaul. Flushing without cascades. In *Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2020.
- [11] M. Bender, M. Farach-Colton, and W. Kuszmaul. Achieving optimal backlog in multi-processor cup games. In *Proceedings of the 51st Annual ACM Symposium on Theory of Computing (STOC)*, 2019.
- [12] M. A. Bender, J. Christensen, A. Conway, M. Farach-Colton, R. Johnson, and M.-T. Tsai. Optimal ball recycling. In *Proc. 30th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 2527–2546, San Diego, CA, January 2019.
- [13] P. Damaschke and Z. Zhou. On queuing lengths in on-line switching. *Theoretical computer science*, 339(2-3):333–343, 2005.
- [14] P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing (STOC)*, pages 365–372, 1987.
- [15] P. F. Dietz and R. Raman. Persistence, amortization and randomization. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 78–88, 1991.
- [16] J. Fischer and P. Gawrychowski. Alphabet-dependent string searching with wexponential search trees. In *Annual Symposium on Combinatorial Pattern Matching (CPM)*, pages 160–171, 2015.
- [17] R. Fleischer and H. Koga. Balanced scheduling toward loss-free packet queuing and delay fairness. *Algorithmica*, 38(2):363–376, Feb 2004.
- [18] H. R. Gail, G. Grover, R. Guérin, S. L. Hantler, Z. Rosberg, and M. Sidi. Buffer size requirements under longest queue first. *Performance Evaluation*, 18(2):133–140, 1993.
- [19] L. Gasieniec, R. Klasing, C. Levcopoulos, A. Lingas, J. Min, and T. Radzik. Bamboo garden trimming problem (perpetual maintenance of machines with different attendance urgency factors). In *International Conference on Current Trends in Theory and Practice of Informatics*, pages 229–240. Springer, 2017.
- [20] M. H. Goldwasser. A survey of buffer management policies for packet switches. *SIGACT News*, 41(1):100–128, 2010.
- [21] M. T. Goodrich and P. Pszona. Streamed graph drawing and the file maintenance problem. In *International Symposium on Graph Drawing*, pages 256–267. Springer, 2013.
- [22] N. Guan and W. Yi. Fixed-priority multiprocessor scheduling: Critical instant, response time and utilization bound. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, pages 2470–2473. IEEE, 2012.
- [23] S. Howard. Double disk failure, 2015.
- [24] T. Kopelowitz. On-line indexing for general alphabets via predecessor queries on subsets of an ordered list. In *Proceedings of the 53rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 283–292, 2012.
- [25] W. Kuszmaul. Achieving optimal backlog in the vanilla multi-processor cup game. In *Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2020.
- [26] A. Litman and S. Moran-Schein. On distributed smooth scheduling. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 76–85, 2005.
- [27] A. Litman and S. Moran-Schein. Smooth scheduling under variable rates or the analog-digital confinement game. *Theor. Comp. Sys.*, 45(2):325–354, June 2009.
- [28] A. Litman and S. Moran-Schein. On centralized smooth scheduling. *Algorithmica*, 60(2):464–480, 2011.
- [29] C. L. Liu. Scheduling algorithms for multiprocessors in a hard real-time environment. *JPL Space Programs Summary*, 1969, 1969.
- [30] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.
- [31] C. McDiarmid. On the method of bounded differences. *Surveys in combinatorics*, 141(1):148–188, 1989.
- [32] M. Moir and S. Ramamurthy. Pfair scheduling of fixed and migrating periodic tasks on multiple resources. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, pages 294–303, 1999.
- [33] C. W. Mortensen. Fully-dynamic two dimensional orthogonal range and line segment intersection reporting in logarithmic time. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 618–627, 2003.
- [34] M. Rosenblum, M. X. Goemans, and V. Tarokh. Universal bounds on buffer size for packetizing fluid policies in input queued, crossbar switches. In *Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, volume 2, pages 1126–1134, 2004.
- [35] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an mttf of 1, 000, 000 hours mean to you? *File and Storage Technologies (FAST)*, 7(1):1–16, 2007.
- [36] A. Vattani, F. Chierichetti, and K. Lowenstein. Optimal probabilistic cache stampede prevention. *Proceedings of the VLDB Endowment*, 8(8):886–897, 2015.

## A Proof of Claim 1

*Proof.* [Proof of Claim 1] Consider any (possibly randomized) adaptive algorithm for Alice, and let  $X$  be the random variable denoting Alice's profit in the game.

For  $\lambda > 0$ , define

$$M_{k,\mu}(\lambda) = \mathbb{E}[e^{\lambda X}]$$

to be the moment generating function of  $X$ . The key claim is that

$$(A.1) \quad M_{k,\mu}(\lambda) \leq e^{(e^\lambda - 1)\mu}.$$

We prove (A.1) by induction on  $k$ . Suppose that (A.1) holds for  $M_{k',\mu}(\lambda')$  for all  $k' < k$ , any  $\lambda' > 0$ , and any adaptive algorithm for Alice; as a base case, (A.1) is immediate for  $k = 0$ . Let  $X_1$  be a random variable for the profit Alice makes from her first bet and  $X' = X - X_1$ . For any value  $p \leq \mu$  that Alice may select for  $p_1$ ,

$$\begin{aligned} \mathbb{E}[e^{\lambda X} \mid p_1 = p] &= pe^\lambda \cdot \mathbb{E}[e^{\lambda X'} \mid X_1 = 1, p_1 = p] \\ &\quad + (1 - p) \cdot \mathbb{E}[e^{\lambda X'} \mid X_1 = 0, p_1 = p]. \end{aligned}$$

By the inductive hypothesis,

$$\mathbb{E}[e^{\lambda X'} \mid X_1 = 1, p_1 = p] \leq e^{(e^\lambda - 1)(\mu - p)}$$

and

$$\mathbb{E}[e^{\lambda X'} \mid X_1 = 0, p_1 = p] \leq e^{(e^\lambda - 1)(\mu - p)}.$$

Thus

$$\mathbb{E}[e^{\lambda X} \mid p_1 = p] \leq (p \cdot e^\lambda + (1 - p)) e^{(e^\lambda - 1)(\mu - p)}.$$

Using the identity,  $1 + x \leq e^x$  with  $x = p \cdot (e^\lambda - 1)$ , it follows that

$$\mathbb{E}[e^{\lambda X} \mid p_1 = p] \leq e^{p(e^\lambda - 1)} e^{(e^\lambda - 1)(\mu - p)} = e^{(e^\lambda - 1)\mu}.$$

Since this holds for all  $p$ , (A.1) follows.

Using (A.1), we can complete the proof of the lemma as follows. By Markov's inequality,

$$\Pr[X > (1 + \delta)\mu] \leq \Pr[e^{\lambda X} > e^{\lambda(1 + \delta)\mu}] \leq \frac{\mathbb{E}[e^{\lambda X}]}{e^{\lambda(1 + \delta)\mu}}.$$

By (A.1), it follows that

$$\Pr[X > (1 + \delta)\mu] \leq \exp((e^\lambda - 1 - \lambda(1 + \delta))\mu).$$

Plugging in  $\lambda = \ln(1 + \delta)$ , which one can show by the derivative test minimizes the expression on the right side, yields

$$\Pr[X > (1 + \delta)\mu] \leq \exp((\delta - \ln(1 + \delta))(1 + \delta)\mu).$$

Using the identity  $\ln(1 + \delta) \geq \frac{\delta}{1 + \delta/2}$ , gives

$$\begin{aligned} \Pr[X > (1 + \delta)\mu] &\leq \exp\left(\delta \cdot \left(1 - \frac{1 + \delta}{1 + \delta/2}\right) \cdot \mu\right) \\ &\leq \exp\left(\delta \cdot \frac{-\delta}{2 + \delta} \cdot \mu\right) \\ &\leq \exp(-\delta^2 \mu/3). \end{aligned}$$

□