Tight Bounds for the Partial-Sums Problem

Mihai Pătraşcu* Erik D. Demaine*

Abstract

We close the gaps between known lower and upper bounds for the online partial-sums problem in the RAM and group models of computation. If elements are chosen from an abstract group, we prove an $\Omega(\lg n)$ lower bound on the number of algebraic operations that must be performed, matching a well-known upper bound. In the RAM model with b-bit memory registers, we consider the well-studied case when the elements of the array can be changed additively by δ-bit integers. We give a RAM algorithm that achieves a running time of $\Theta(1 + \lg n/\lg(b/\delta))$ and prove a matching lower bound in the cell-probe model. Our lower bound is for the amortized complexity, and makes minimal assumptions about the relations between n, b, and δ. The best previous lower bound was $\Omega(\lg n/(\lg\lg n+\lg b))$, and the best previous upper bound matched only in the special case $b = \Theta(\lg n)$ and $\delta = O(\lg\lg n)$.

1 Introduction

The *partial-sums problem* is to maintain an array $A[1..n]$ subject to the following operations:

`update`(k, Δ)**:** modify $A[k] \leftarrow A[k] + \Delta$.
`sum`(k)**:** returns the partial sum $\sum_{i=1}^{k} A[i]$.

Given an order on the elements, we may extend the problem to include the following operation [23]:

`select`(σ)**:** returns an index i satisfying $\text{sum}(i-1) < \sigma \leq \text{sum}(i)$. To guarantee the uniqueness of the answer, we require that $A[i] > 0$ for all i.

The minimal setting in which the partial-sums problem can be considered is the *semigroup model.* Here elements are chosen from an abstract semigroup, and the running time of an algorithm is the number of algebraic operations (i.e., additions) it performs. A similar model is the *group model* of computation, in which subtraction is also available. A quite different setting for this problem is the *transdichotomous RAM model*, where the elements of the array are integers that each fit in a machine register. In this model, we define two additional parameters: b, the number of bits in a word; and δ, the number of bits needed to represent an argument Δ to `update` in the standard two's complement form. Naturally, $\delta \leq b$, and we use the standard assumption that $b \geq \lg n$.[1] Under the usual transdichotomous assumption, we consider b and δ arbitrary parameters which may grow with n.

Our treatment of the RAM model leads to a slightly different version of the problem, that of computing partial sums in $\mathbb{Z}/m\mathbb{Z}$ on a b-bit RAM. This version is not so natural, and has fewer applications than the one described above, but is theoretically cleaner. We use this modified problem (for $m = 2^\delta$) in our lower-bound argument. Furthermore, our upper bound extends easily to this case, giving a tight bound of $\Theta(1 + \lg n/\lg(b/\lg m))$ for this problem.

Table 1 gives a brief summary of main results for the partial-sums problem, both old and new. Details follow below.

1.1 Our Results. In Section 2 we give an improved data structure that achieves a running time of $O(1 + \lg n/\lg(b/\delta))$ per operation. Our key contribution is to recognize that the precomputed tables used by [10] and [23] can be replaced by careful use of word-level parallelism techniques. This makes it possible to obtain an upper bound that applies naturally to various combinations of n, b, and δ, while keeping the space $O(n)$. Note, for instance, that our bound is identical to the classic logarithmic bound for $\delta = \Theta(b)$, and achieves a running time of $O(\lg n/\lg\lg n)$ for $\delta = O(b/\lg^\varepsilon n)$, for any constant $\varepsilon > 0$. This is the same running time as the data structure given by [10], but under lighter assumptions. We also show that our data structure can support the `select` operation in the same running time as `sum`, improving the results of [23].

Subsequent sections concern our lower bound work. We develop a technique for applying Kolmogorov complexity to dynamic data structure problems, that we believe will find applications beyond the partial-sums problem. The general framework of our proofs is described in more detail in Section 3. Using these ideas, we give a simple proof of our logarithmic lower bound for the group model in Section 4. Our bound is amor-

*MIT Laboratory for Computer Science, 200 Technology Square, Cambridge, MA 02139, USA, {mip, edemaine}@mit.edu

[1]We use the following notation: $\lg n = \lceil \log_2(2 + n) \rceil$.

Online partial-sums	Upper bounds		Lower bounds	
Semigroup model	$\boldsymbol{O(\lg n)}$	[classic]	$\Omega(\lg n/\lg\lg n)$	[24]
			$\boldsymbol{\Omega(\lg n)}$	[17]
Group model	$\boldsymbol{O(\lg n)}$	[classic]	$\Omega(\lg n/\lg\lg n)$	[15]
			$\boldsymbol{\Omega(\lg n)}$	[NEW]
RAM model	$O(\lg n/\lg\lg n)$ for $\delta = O(\lg\lg n)$	[10]	$\Omega(\lg n/(\lg\lg n + \lg b))$	[15]
	$\boldsymbol{O(\lg n/\lg(b/\delta))}$	[NEW]	$\boldsymbol{\Omega(\lg n/\lg(b/\delta))}$	[NEW]

Table 1: Summary of old and new results on partial sums. Optimal results are bold.

tized and is derived by considering the average case of a natural probability distribution. The proof also gives a powerful characterization of the hardness of an instance of the problem. Using this insight, we can construct a fixed sequence of operations that takes $\Omega(\lg n)$ amortized time per operation for any algorithm. The algorithm is allowed to know the indices touched by `update` and `sum` beforehand, but does not know the parameters Δ to `update`. In the semigroup model, we can eliminate even this restriction, and make our bound hold for the totally offline case, which gives an independent proof of the results of [17].

In Section 5, we give lower bounds on the cell-probe complexity of the partial-sums problem, which immediately apply to the RAM model as well. We first treat the case when $\delta = \Theta(b)$ (no essential restriction is imposed on δ), and give a tight $\Omega(\lg n)$ lower bound on the cell-probe complexity. We then show the data structure proposed in Section 2 is asymptotically optimal for any combination of n, b and δ. This proof requires additional insight, and depends crucially on the algorithm not knowing the indices touched by `update` and `sum` in the future. As before, our bound is amortized, and is derived by considering the average case of a natural probability distribution. When $\delta = 1$, our bound is equivalent to the bound given by [15], under our original assumption that $b = \Omega(\lg n)$.

Our results also translate easily to the external-memory and cache-oblivious models. If B is the number of words in a memory-transfer block, our lower bound is $\Omega\left(\frac{\lg n}{\lg B + \lg(b/\delta)}\right)$. A matching upper bound can be obtained cache-obliviously by combining our data structure with an optimal tree layout [1]. This transference is made possible only by the elimination of precomputed tables. An easier $O(\log_B n)$ upper bound can be obtained by combining the classic binary tree solution with a van Emde Boas layout [7].

1.2 Previous Work. The partial-sums problem is a classic problem and has been studied rather extensively. Besides being a natural range-query and data-structure problem, it has applications to list indexing and dynamic ranking [10], dynamic arrays [23], and arithmetic coding [11]. The problem is also well-studied on the lower-bound side; it is interesting to note (see [22]) that many techniques for proving lower bounds were developed initially for the partial-sums problem, which is rivaled in this respect only by the predecessor problem.

It is easy to obtain an $O(\lg n)$ upper bound for all three operations using a balanced binary tree with the elements of A in the leaves, and storing partial sums for each subtree. A simple variation of this scheme yields an *implicit* data structure occupying only n memory locations [11].

For the semigroup model, this logarithmic bound is the best possible, even if the problem is offline, as shown in [17]. Previously, a lower bound of $\Omega(\lg n/\lg\lg n)$ had been derived in [24]. A tight logarithmic lower bound for a slightly harder problem appears in [13]. Finally, a trade-off between the running times of the operations was given by [8]; this tradeoff is optimal for a certain class of algorithms [8, 14].

For the group model, the best previous lower bound of $\Omega(\lg n/\lg\lg n)$ is by Fredman and Saks [15]. A tight logarithmic bound (including the lead constant), was given by [14] for the class of "oblivious" algorithms, whose behavior can be described by matrix multiplication. For the offline problem, Chazelle [9] gives the only known lower bound of $\Omega(\lg\lg n)$ per operation. While this offline bound is an important result, it is exponentially weaker than the best known upper bound; closing this gap remains an interesting open problem.

The investigation of the problem in less-restrictive models began with the classic result of Fredman and Saks [15], who developed the chronogram technique and used it to prove a lower bound of $\Omega(\lg n/(\lg\lg n + \lg b))$ on the cell-probe complexity for any $\delta \geq 1$. Their bound is partially matched by a RAM data structure of Dietz [10], which achieves $O(\lg n/\lg\lg n)$ running times provided that $\delta = O(\lg\lg n)$. Unfortunately, these results offer only a piece of the puzzle, because neither scales well with δ; in addition, the upper bound of [10] does not depend of b, and is optimal only if $b = \text{poly}\lg n$.

This incomplete story lead Hampapuram and Fredman [17] to highlight less-restrictive models of computation (namely the group and RAM model) as underdeveloped. Miltersen [22] lists the problem as a challenge for future research and suggests that it might be easier to obtain just a *strongly transdichotomous* lower bound of $\Omega(\lg n)$, matching the simple upper bound mentioned above. (Such a bound need only apply to one worst-case combination of b and δ for every given n.) Our result applies to any combination of n, b, and δ, and immediately gives a strongly transdichotomous bound for $\delta = \Theta(b)$. (Note that this gives a lower bound of $\Omega(\lg n)$ regardless of the relation between b and n.)

Though the bounds of [15] and [10] have not been improved, some progress has been reported in other directions. Ben-Amram and Galil [5] reproved the lower bound of [15] in a more formalized framework, centered around the concepts of problem and output variability. Using these ideas, they were able to show in [6] that the lower bound holds even if the machine registers have infinite precision, but the set of operations is restricted. Husfeldt and Rauhe [19] show that the lower bound of [15] holds even for the promise version of the problem, in which the algorithm is told the requested sum to a ± 1 precision. Several lower bounds for various problems are shown in [20] and [19] by reductions to this result or the original result of [15]. Finally, [2] generalizes the predecessor and partial-sums problems to trees. In this case, values are stored in nodes and along edges, and the `sum` operation reports the sum of the values along a path between two nodes. Their upper bound uses Dietz's data structure and inherits its limitations.

On the upper bound side, Raman et al. [23] improve on the work of Dietz, by showing how to support `select` in $O(\lg n/\lg\lg n)$ if $\delta = O(\lg\lg n)$, and offering some tradeoffs between the running times of the operations. They are also concerned with memory consumption, and show how to reduce the space to $kn + o(kn)$ bits, if the elements of the array are k-bit integers (for some $k \le \lg n$). Recently, Hon et al. [18] gave an additional tradeoff between the running times of `sum` and `update`, while keeping the data structure succinct. They also noted that the lower bounds for the predecessor problem established by Beame and Fich [4] hold for `select`, and showed that these bounds can be matched if one is willing to pay a cost of $\Theta(n^{\varepsilon})$ per `update`.

A new direction for research was opened by [12]. They consider a version of the problem in which the elements of the array are chosen from a fixed monoid (independent of n) known to the algorithm. The dynamic-prefix problem asks to support partial-sum queries; a *dynamic-word* problem asks only for the sum of all elements in the array. Depending on the algebraic properties of the monoid, several bounds are derived for the cell- and bit-probe complexity. We note that the bit-probe complexity for the natural $\mathbb{Z}/2\mathbb{Z}$ case had already been considered by Fredman [14], who proved a lower bound of $\Omega(\lg n/\lg\lg n)$. Closing the gap between this and the known logarithmic upper bound remains an open problem.

2 Upper Bounds

2.1 Partial sums in small arrays.

We begin by supporting `sum` and `update` in constant time if n is "sufficiently small". We will conceptually maintain an array of partial sums $S[1..n]$ defined by $S[k] = \sum_{i=1}^{k} A[i]$. To make it possible to support `update` in constant time, we maintain the array as two separate components, $B[1..n]$ and $C[1..n]$, such that $S[i] = B[i] + C[i]$. The array B will hold values of S that were valid at some point in the past, while more recent updates are reflected only in C. We can use Dietz's incremental rebuilding scheme to maintain every element of B relatively up-to-date: on the t-th `update`, we set $B[t \bmod n] \leftarrow B[t \bmod n] + C[t \bmod n]$ and $C[t \bmod n] \leftarrow 0$. This scheme guarantees that every element in C is affected by at most n updates, and thus is bounded in absolute value by $n \cdot 2^{\delta}$.

The key optimization is to recognize that C can be packed in a machine word. We pack the array in a word by representing each element in the array by a certain range of the bits from the word, with one zero bit of padding between elements. Elements in C can also be negative; in this case, each value will be represented in the standard two's complement form on its corresponding range of bits. Because each element in C can be represented by $O(\delta + \lg n)$ bits, we can pack the whole array C in one word if $n = O(\min\{b/\lg b, b/\delta\})$, for an appropriate choice of constants. We can read and write elements of the array in packed form using a constant number of standard RAM operations (bitwise boolean operations and shift operations).

One consequence of packing C is that our data structure requires only $n + O(1)$ memory locations. A more important consequence is that we can add a given value to all elements $C[i]$, $i \ge k$, in constant time; this operation is the crux of `update`. Refer to Figure 1. First, we create a word with the value to be added appearing in all positions corresponding to the elements of C that need to be changed. We can compute this word using a multiplication by an appropriate binary pattern. The result is then added to the packed representation of C; all the needed additions are performed in one step, using word-level parallelism. Because we are representing negative quantities in two's complement, additions may carry over, and set the padding bits between elements; we therefore force these buffer bits to zero using a bitwise `and` with an appropriate constant mask. We obtain the following result:

$C[4]$	0	$C[3]$	0	$C[2]$	0	$C[1]$	0	$C[0]$	old packed representation of C
00001	0	00001	0	00001	0	00001	0	00001	constant pattern
00001	0	00001	0	00001	0	00000	0	00000	shift right and back left by the same amount
								Δ	argument given to `update`
Δ	0	Δ	0	Δ	0	00000	0	00000	multiply the last two values
$C'[4]$	?	$C'[3]$	?	$C'[2]$	?	$C[1]$	?	$C[0]$	add to the packed representation of C
11111	0	11111	0	11111	0	11111	0	11111	constant cleaning pattern
$C'[4]$	0	$C'[3]$	0	$C'[2]$	0	$C'[1]$	0	$C'[0]$	final value of C, obtained through bitwise `and`

Figure 1: Performing `update`$(2, \Delta)$ at the word level. Here C has 5 elements, each 5 bits long.

Lemma 2.1. *If $n = O(\min\{b/\lg b, b/\delta\})$, we can support* `update` *and* `sum` *on an array of size n in $O(1)$ worst-case running time. The data structure occupies $n + O(1)$ memory locations.*

2.2 Selecting in small arrays. To support `select`, we use a classic result of Fredman and Willard [16] that forms the basis of their fusion-tree data structure. Their result gives us the following black-box functionality: for $n = O(b^{1/5})$,[2] we can construct a data structure that can answer successor queries on a static array of n integers in constant time. The data structure can be constructed in $O(n^4)$ time. As demonstrated in [3], the lookup tables used by the original data structure can be eliminated, if we perform a second query in the sketch representation of the array. The memory consumption of the data structure then becomes $n + O(1)$ words.

As in the previous section, we will maintain the two arrays B and C. We will also store a fusion structure that can answer successor queries in B. Because the fusion structure is static,[3] we abandon the incremental rebuilding of B, in favor of periodic global rebuilding. By a standard de-amortization of global rebuilding, we can then obtain worst-case bounds; a careful implementation also requires only $O(1)$ additional words of space. Our strategy is to rebuild the data structure completely every n^4 operations: we set $B[i] \leftarrow B[i] + C[i]$ and $C[i] \leftarrow 0$, for all i, and rebuild the fusion structure over B. While servicing a `select` that doesn't occur immediately after a rebuild, the successor in B found by the fusion structure might not be the appropriate answer to the `select` query, because of recent updates. We will describe shortly how the correct answer can be computed by also examining the array C; the key realization is that the real successor must be close to the successor in B in terms of their partial sums.

Central to our solution is the way we rebuild the data structure every n^4 operations. We begin by splitting S into runs of elements satisfying $S[i+1] - S[i] < n^4 \cdot 2^\delta$; recall that we assumed $S[i] < S[i+1]$ for the `select` problem. We denote by $rep(i)$ the first element of the run containing i (the representative of the run); also let $len(i)$ be the length of the run containing i. Each of these arrays can be packed in a word, because we already limited ourselves to $n = O(b^{1/5})$. Finally, we let every $B[i] \leftarrow B[rep(i)]$ and $C[i] \leftarrow S[i] - B[rep(i)]$. Conveniently, C can still be packed in a word. Indeed, the value stored in an element after a rebuild is at most $n \cdot \left(n^4 \cdot 2^\delta\right)$, and it can subsequently change by less than $n^4 \cdot 2^\delta$. Therefore, it takes $O(\lg n + \delta)$ bits to represent an element of C, so we only need to impose the condition that $n = O(\min\{b/\delta, b^{1/5}\})$.

It remains to show how a *select*(σ) query can be answered. Let k denote the successor in B identified by the fusion-tree data structure; k satisfies $B[k-1] < \sigma \leq B[k]$. We know that k is the representative of a run, because all elements of a run have equal values in B. By construction, runs are separated by gaps of at least $n^4 \cdot 2^\delta$; because such gaps cannot be closed by n^4 updates, we obtain a restriction on the correct answer to the select query. More precisely, the answer must be either an index in the run starting at k, or an index in the run ending at $k-1$, or exactly equal to $k + len(k)$. We can distinguish between these cases in constant time, using two calls to `sum` followed by comparisons. If we have identified the correct answer as exactly $k + len(k)$, we are done.

Otherwise, let us assume that the answer must be an index in the run starting at k. Because elements of a run have equal values of B, our task is to identify the unique index i in the run satisfying $C[i-1] < \sigma - B[k] \leq C[i]$. We claim that we can employ word-level parallelism to compare all elements in C with $\sigma - B[k]$. This is similar to a problem discussed in [16], but we must also handle negative quantities. The solution is to subtract $\sigma - B[k]$ in parallel from all elements in C; if elements of C are oversized by 1 bit, we can avoid

[2]The original paper restricted n to $O(b^{1/6})$; however, a careful analysis reveals that $O(b^{1/5})$ is enough. The exact power is usually irrelevant, because it only translates into a constant factor in the running time.

[3]Q-heaps offer a dynamic alternative to fusion trees. However, they require large precomputed tables, which make it impossible to simultaneously obtain time bounds in b and space bounds in n.

overflow. The sign bits of every element then give the results of the comparisons. The answer to the query can be found by summing up the sign bits corresponding to elements in our run, which indicates how many elements in the run were smaller than $\sigma - B[k]$. Because these bits are sufficiently sparse, we can sum them up using a multiplication with a constant pattern, as described in [16]. We have thus achieved the following result:

Lemma 2.2. *If $n = O(\min\{b/\delta, b^{1/5}\})$, we can support* `update`*,* `sum`*, and* `select` *on an array of size n in constant time. The data structure occupies $n + O(1)$ memory locations.*

2.3 The data structure. We use the approach described in [10] to construct a solution for arbitrary n based on our improved solutions for small arrays. Consider a static balanced tree with branching factor B with the elements of the array $A[1..n]$ in the leaves. Let the weight of a node be defined as the sum of the elements of A stored in the leaves of its subtree. Each node will hold a partial-sums data structure of size B with the weights of its children. All three operations in the large data structure translate into a sequence of operations on the small data structures of the nodes along a particular root-to-leaf path.

If $B = \Theta(\min\{b/\lg b, b/\delta\})$, we can support `update` and `sum` in the small data structure of each node in constant time. The running time of these operations in the large data structure is then $O(\log_B n) = O(\lg n/\lg B) = O\left(\max\left\{\frac{\lg n}{\lg(b/\lg b)}, \frac{\lg n}{\lg(b/\delta)}\right\}\right) = O(\lg n/\lg(b/\delta))$. In this solution we assumed $2 \le B \le n$. If $B = \Theta(1)$, we do not obtain an asymptotic advantage by packing multiple values in a word; in this case we can instead apply the solution of [11] for a running time of $O(\lg n)$. In the second extreme case, $B = \omega(n)$, we obtain $O(1)$ running times by directly applying Lemma 2.1. These cases cover all possible asymptotic relations between (n, b, δ), and $O(1 + \lg n/\lg(b/\delta))$ is an accurate characterization of our upper bound in all cases. Our data structure also occupies only $n + o(n)$ memory locations in all three cases.

Similar considerations apply if we want to support `select` as well. In particular, B may decrease polynomially, but this only affects constant factors in the time bounds. Thus we obtain the following result:

Theorem 2.1. *We can support* `update`*,* `sum`*, and* `select` *over an array of size n in $O(1 + \lg n/\lg(b/\delta))$ worst-case running time. The data structure requires $n + o(n)$ memory locations.*

3 Framework for Lower Bounds

3.1 Overview. For our lower bounds discussion, we concentrate on sequences of exactly n pairs of operations. Each pair contains an `update` and a `sum` operation on the same element of the array. Furthermore, the elements probed by the n pairs form a permutation of the elements of the array. It follows that we can characterize the sequence of operations by a permutation $\pi(i)$, $i = 1..n$, and an array of update values $\Delta[1..n]$. We associate each pair of operations with the interval of time that the algorithm spends serving it. Using this terminology, we say that at time i the algorithm is serving the pair of operations: update$(\pi(i), \Delta[i])$; sum$(\pi(i))$. Let $S[i]$ denote the answer returned by the `sum` query executing at time i. Thus, $S[k] = \sum\{\Delta[i] : (i \le k) \text{ and } (\pi(i) \le \pi(k))\}$.

Before we continue, we make one technical point. It is often desirable for an amortized lower bound to apply to an arbitrarily long sequence of operations. In what follows we concentrate on sequences of n operations for the sake of clarity. However, our arguments extend easily to sequences of arbitrary length, formed by concatenating different sequences of length n as defined above.

As explained above, we divide the time axis into n intervals such that the algorithm is servicing the ith pair of operations for the duration of interval i. We prove our lower bounds in the cell-probe model, so we ignore all computation and concentrate only on memory accesses. Furthermore, we choose to ignore write instructions completely. Finally, we ignore read instructions from memory locations written during the same time interval. The rest of the read instructions can be characterized by a pair (t_w, t_r) denoting the time t_w when the memory location was written, and the time t_r when the read occurs; in particular, $t_w < t_r$. We sometimes refer to such a read instruction as a *communication* between time t_w and time t_r.

Our technique involves conceptually constructing a balanced tree of a certain degree B over the time axis, so that each leaf is an interval of time associated with an `update-sum` pair. For each node in the tree, we establish a lower bound on the amount of communication that must occur strictly between the subtrees of its children. It should be clear that this approach does not double-count any read instruction. Indeed, we associate such a read instruction with a unique node, determined by the longest common prefix of (t_w, t_r) considered in base B. In the following sections, we show lower bounds on the communication associated with some arbitrary node. The reason we can simply sum these lower bounds for all nodes in the tree is that out bounds are for the average case, and we can invoke linearity of expectation.

3.2 An encoding scheme. We arrive at the communication lower bounds for each node by analyzing an encoding scheme. Let us fix an arbitrary node, and number its children from left to right, i.e., in the increasing direction of the time axis. Let s_k denote the earliest moment in time contained in the subtree of child k, and let f_k denote the latest such moment; thus, $s_k = f_{k-1} + 1$. In the following sections, we establish lower bounds on the communication between the subtree of a child k and the subtrees of children before k. To do so, we imagine a situation in which we know

1. the entire permutation $\pi(i)$;
2. part of the sequence of updates, namely, $\Delta[i]$ for any $i \notin \{s_1..f_{k-1}\}$, i.e., all the values of Δ outside the subtrees of the left siblings of k;
3. for each read instruction (t_w, t_r) with $s_1 \le t_w \le f_{k-1}$ and $s_k \le t_r \le f_k$, the address and the contents of the read memory location. Note that these are exactly the read instructions we are trying to bound.

A crucial point is that we can determine $S[s_k..f_k]$ given this information. To do that, we begin by running the algorithm for the time period $1..(s_1 - 1)$; this is possible because we know $\pi(i)$ and $\Delta[i]$ for $i < s_1$. We then skip the time period $s_1..f_{k-1}$ and run the algorithm for the time period $s_k..f_k$ to generate the partial sums $S[s_k..f_k]$. To see why this is possible, notice that a read instruction issued during time period $s_k..f_k$ falls into one of three categories:

$t_w \ge s_k$**:** We can recognize this case by maintaining a list of memory locations written during the simulation; the data is immediately available.

$s_1 \le t_w < s_k$**:** The contents of the memory location is available in our knowledge base; we can recognize this case by looking at the list of memory addresses mentioned in item 3 above.

$t_w < s_1$**:** The contents of the cell is determined from the state of the memory upon finishing the first simulation up to time s_1.

It remains to characterize the information that we can extract about $\Delta[s_1..f_{k-1}]$ given $S[s_k..f_k]$ and the rest of the values of Δ. Let $P = \{\pi(i) : s_1 \le i \le f_{k-1}\}$ denote the set of elements updated during the time $s_1..f_{k-1}$ and, similarly, let $Q = \{\pi(i) : s_k \le i \le f_k\}$. Roughly, the amount of information we can extract about $\Delta[s_1..f_{k-1}]$ is given by the number of runs of elements from P in the ordered listing of $P \cup Q$. More formally, we can express P as the union of disjoint sets $P_1, \dots, P_l$, and similarly Q as the union of $Q_1, \dots, Q_l$, such that $\max(Q_1) < \min(P_1) \le \max(P_1) < \min(Q_2) \le \max(Q_2) < \min(P_3) \le \cdots$, where we allow Q_1 and P_l to be empty. We call l the *interleaving factor* between P and Q. The following result justifies the importance of l, and summarizes the discussion so far:

LEMMA 3.1. *Given the information in items 1–3 above, we can extract $l - 1$ linearly independent sums of the update values $\Delta[s_1..f_{k-1}]$, where l is the interleaving factor defined above.*

From the definition of the partial sums, it is easy to see that we can determine $\sum\{\Delta[i] : \pi(i) \in P_k\}$ for all $k = 1..(l-1)$. Because the P_k's are disjoint, the sums are linearly independent. It remains to quantify the interleaving factor l:

LEMMA 3.2. *Let π be a permutation of order n chosen uniformly at random. Let $P = \{\pi(i) : a_1 \le i \le b_1\}$ and $Q = \{\pi(i) : a_2 \le i \le b_2\}$ with $b_1 < a_2$ (i.e., disjoint segments of the permutation). Then the expected value of the interleaving factor of P and Q is asymptotically $\Theta(\min\{|P|, |Q|\})$.*

The proof is elementary by linearity of expectation.

4 Lower Bounds for the Group Model

We are now ready to establish an amortized lower bound of $\Omega(\lg N)$ in the group model. We choose the permutation π uniformly at random. The values of the array Δ are irrelevant, because in this model the algorithm may not examine the value stored in a memory location, but can only perform algebraic operations with it. We begin the analysis by constructing a balanced binary tree over the time axis, as described in the framework discussion.

Let us now fix a node in the tree, and let L denote the number of leaves in its left subtree, i.e., the number of data structure operations performed during the time covered by the subtree. We are trying to bound the number R of read instructions executed in the right subtree of the node that read memory locations last written in the left subtree. Condition on a fixed permutation π, such that the interleaving factor between the left and right subtrees is l. Assume we know the $n - L$ values of Δ outside the left subtree, and the data given by the R reads that constitute the communication between the left and right subtrees. Lemma 3.1 guarantees that we can reconstruct $l - 1$ independent sums of the L unknown values from Δ. We can therefore construct a linear operator that takes the known values to the $l - 1$ independent sums from the left subtree and the $n - L$ values of Δ outside the left subtree. The image of the operator has dimension $n - L + l - 1$. It follows that its domain must have dimension at least $n - L + l - 1$. But the dimension of the domain is at most $n - L + R$, the number of values known initially. It follows that $R \geq l - 1$. Taking the expected value for a random permutation π, and applying Lemma 3.2, we get $E[R] = \Omega(L)$.

Solving an elementary recurrence relation, we obtain that the total number of read instructions performed during the entire execution is bounded by $\Omega(n \lg n)$ in the average case. But the number of read instructions is equal to the number of algebraic operations up to constant factors, which establishes our result.

Our proof technique also gives a deterministic lower bound on the hardness of a particular sequence of operations: the sum of the interleaving factors for all nodes in the tree. Using this insight we can construct fixed sequences of operations which take $\Omega(n \lg n)$ for any algorithm. We describe one such permutation, the *perfect shuffle*. We construct the permutation of order $2n$ recursively, by shuffling two copies of the permutation of order n. Formally, if π is the permutation of order n, we construct a permutation of order $2n$ as $\pi'(i) = 2 \cdot \pi(i) - 1$; $\pi'(i + n) = 2 \cdot \pi(i)$, for $i = 1..n$. The proof that this sequence of operations takes $\Omega(n \lg n)$ time for any algorithm is very similar to the argument above.

This fixed hard instance is still an online problem: $S[i]$ must be computed before $\Delta[i + 1]$ is available. Our argument also works for the totally offline problem in the semigroup model, which gives a new proof of the results of [17]. The reason we can analyze the offline problem in the semigroup model is that the lack of a subtraction operation makes the information flow "unidirectional", similar to the online problem. In this case, the leaves of our tree are the elements of the array Δ, in order. We associate any computation with the highest index i such that $\Delta[i]$ appears in the expression of the result, which is a linear combination of the elements of Δ. Because subtraction is not available, a value cannot be cancelled out from an expression, so computation moves only from left to right in our tree.

Theorem 4.1. *Any algorithm for the online partial-sums problem in the group model has an amortized running time per operation of $\Omega(\lg n)$, in the average case of a certain probability distribution. Furthermore, there exists a fixed sequence of operations that requires $\Omega(\lg n)$ amortized time per operation for any algorithm.*

5 Cell-Probe Lower Bounds

5.1 A first bound. In this subsection, we establish a weaker version of our lower bound in the cell-probe model, by proving a bound of $\Omega\left(\lg n \cdot \frac{\delta}{b}\right)$. This bound is tight only if $\delta = \Theta(b)$, that is, if no essential restriction is imposed on the update values. However, this simpler case will help make clear the framework for our work in the cell-probe model, and suffices to give a strongly transdichotomous lower bound of $\Omega(\lg n)$.

For technical reasons, we impose a standard restriction that the number of memory locations is at most 2^b. This restriction is satisfied naturally if a pointer fits in a word, as in the RAM model. All proofs work with a weaker $2^{O(b)}$ restriction after an adjustment of constants. As mentioned in the introduction, we concentrate on the partial-sums problem modulo 2^δ; thus we can think of the updates $\Delta[i]$ as unsigned δ-bit integers, by using the two's complement transformation. A solution to the original problem also gives a solution to this (easier) problem, as long as we can avoid overflow; for this reason, we impose another technical condition, that $\delta + \lg n < b$. Because we already have $\lg n \leq b$ and $\delta \leq b$, this restriction is easily seen to be irrelevant, save for constant factors. Finally, as discussed in Section 3, if we want arbitrarily long sequences of operations, we can concatenate sequences of order n, interspersed by sequences of n updates that bring the array back to zero.

The sequence of operations that we analyze is constructed by choosing the permutation π uniformly at random, and choosing $\Delta[i]$ independently and uniformly from $0..(2^\delta - 1)$. As before, we construct a balanced binary tree over the time axis and, for every node in the tree, we bound from below the number of read instruction performed in the right subtree that access data written in the left subtree. In this section we obtain an average-case lower bound of $\Omega\left(L \cdot \frac{\delta}{b}\right)$, where L is the number of leaves in the left subtree of the node. In aggregate, we obtain a bound of $\Omega\left(n \lg n \cdot \frac{\delta}{b}\right)$ on the average number of read instructions performed while servicing the entire sequence of operations.

Our analysis for a fixed node in the tree makes use of a simple result from Kolmogorov complexity. Let $A[1..n]$ be an array of integers, chosen independently and uniformly at random from $0..(2^\delta - 1)$. Then any encoding scheme requires $n \cdot \delta - O(1)$ bits in the average case to encode this array. For further details on Kolmogorov complexity, the reader is referred to [21].

Now we are ready to analyze the communication complexity between the subtrees of an arbitrary node with L leaves in the left subtree. Similar to the group model, we begin by conditioning on a permutation π, such that the interleaving factor associated with our node is l. We then condition on some arbitrary values of Δ outside the left subtree of our node. Let us now assume that we know the communication between the left and right subtrees (R read instructions), which can be represented by $R \cdot 2b$ bits, the address and the content of each memory location that is probed. From the framework discussion, we know that we can extract $l-1$ linearly independent sums of values from the unknown segment of Δ. Because the permutation is fixed, so is the structure of these sums (which $\Delta[i]$ contributes to which sum). It is trivial to prove that these sums, taken modulo 2^δ, are independent random variables uniformly distributed in $0..(2^\delta - 1)$. By Kolmogorov complexity, any encoding for this array requires $(l-1)\delta - O(1)$ bits on average. But we were able to encode this information using $E[R] \cdot 2b$ bits on average. (The expectation of R is needed because the algorithm may behave differently depending on the values it reads.) It follows that $E[R] = \Omega\left(l \cdot \frac{\delta}{b}\right)$. We conclude by taking the expectation over the values of Δ outside the left subtree and over the permutation π, because we began by conditioning on arbitrary choices for these values. Applying Lemma 3.2 to bound l, we obtain $E[R] = \Omega\left(L \cdot \frac{\delta}{b}\right)$, which concludes our proof.

5.2 A tight bound for all cases. We first note that the analysis from the previous section gives a tight bound on the number of *bits* that must be communicated: $\Omega(n \lg n \cdot \delta)$. Given that we can pack b bits in a word, it is straightforward to conclude that $\Omega\left(n \lg n \cdot \frac{\delta}{b}\right)$ read instructions must be performed. Our strategy for improving this bound is to argue that an algorithm cannot make efficient use of all b bits of a word, if future queries are sufficiently unpredictable.

We begin by constructing a tree over the time axis with a branching factor of $B = b/\delta$. The case $B = \Theta(1)$ is covered by our analysis in the previous section; here we concentrate on the case when B is super-constant. Our tree must be balanced in the sense that the number of leaves in a subtree must be at most twice the number of leaves in a sibling subtree. We then choose the permutation π and the array Δ randomly as before.

For a fixed node in the tree, we split the children of the node into three approximately equal parts. We ignore the leftmost third of the children, because there is too little information they need to learn from their left siblings. (Recall that "left" means backwards on the time axis.) We also ignore the rightmost third, because queries can become too predictable. For a child in the middle third, we prove that the number of read instructions executed in the subtree of that child, accessing data written in a subtree of some left sibling, must be $\Omega(L)$ on average, where L is the number of leaves in the subtree of the child. Simple computations then show that the total number of read instructions must be $\Omega(n \log_B n)$, which is the desired result.

For a fixed child in the middle third, let t_1 denote the first moment in time in the subtree of its leftmost sibling, let t_2 denote the first moment in time from the subtree of our fixed child, and let t_3 denote the last moment in time in the subtree of the rightmost sibling. We begin by conditioning on some arbitrary values of Δ outside $t_1..(t_2-1)$; we also condition on some arbitrary values of $\pi(i)$ for $i \notin [t_2..t_3]$. Note that this determines the values of $\pi(t_2..t_3)$, but does not determine their order. Let l be the interleaving factor of $\pi(t_1..t_2-1)$ and $\pi(t_2..t_3)$. By Lemma 3.2, $E[l] = \Theta(B \cdot L)$; also $l \le \Theta(B \cdot L)$. From these facts, it follows quite easily that there exists a constant c such that $l \ge c \cdot BL$ with $\Theta(1)$ probability. The analysis below is for the case $l \ge c \cdot BL$; because this event happens with constant probability, this restriction only affects constant factors in our lower bounds.

Between t_2 and t_3, the algorithm will have to determine l linearly independent sums of the values $\Delta[t_1..(t_2-1)]$ through accesses to memory cells written between t_1 and $t_2 - 1$. The structure of these sums is fixed given our conditions on π. Our lower bound will stem from an encoding scheme for these sums and for the values $\pi(t_2..t_2+L-1)$. (Note that the sums and values of π are independent random quantities.) Applying Kolmogorov complexity, it follows easily that any such encoding requires $l \cdot \delta + L \lg(BL) - O(L)$ bits on average, which is $\Theta(L)$. Informally, we argue that if the algorithm has read too little (less than a constant fraction) of this information before a query, then in order to answer the query, it will have to perform, with constant probability, at least one read instruction accessing data written between t_1 and $t_2 - 1$.

Formally, let R_t, $t \in [t_2..(t_2+L-1)]$, be defined as follows. If the algorithm performs at least $(c/4) \cdot L$ read instructions between t_2 and $t_2 + L - 1$, accessing data written between t_1 and $t_2 - 1$, we distribute $(c/4) \cdot L$ balls in L bins at random, and let R_t be the number of balls in bin $t - t_2$. Otherwise, we let R_t be the number of read instructions performed at time t accessing data written between t_1 and $t_2 - 1$. Note that $\sum R_t$ never exceeds the number of read instructions we are trying to bound. We will show that $E[R_t] \ge c/8$, which implies that $E[\sum R_t] = \Omega(L)$.

Let us assume $E[R_{t_0}] \le c/8$ for some t_0. We now form an encoding for the quantities described above. We first encode the elements $\pi(t_2..(t_2+L-1))$; this is trivial to do in $L \lg(BL) + O(L)$ bits, because the elements of $\pi(t_2..t_3)$ are known, except for their order. It remains to encode the l linearly independent sums. We choose from two schemes, based on circumstances determined by the values of the random variables; an extra bit encodes which scheme we are using. If $\sum R_t > (c/4) \cdot L$, our encoding is simply the array of sums, occupying $l \cdot \delta$ bits. Conditioned on being in this case, $E[R_{t_0}] = c/4$, so the probability that we are in this case is at most

1/2. Furthermore, conditioned on not being in this case, $E[R_{t_0}]$ must still be less than $c/8$. In this second case, we use the following encoding scheme. First we encode the read instructions executed before time t_0, accessing data written between t_1 and $t_2 - 1$; it takes at most $(c/4) \cdot L \cdot 2b$ bits to encode the content and address of each probed location. Then, for each partial sum that cannot be determined exclusively based on this information, we encode the actual sum. By encoding these in order, we do not have to actually specify which these sums are. It is not hard to show that we can actually reconstruct the partial sums by simulating the algorithm, but note that we really need $\pi(t_2..t_0)$ as part of the encoding. Because $E[R_{t_0}] \le c/8$, the number of partial sums that we cannot determine based on previous reads is small on average, namely, at most $(c/4)BL$, because there are at most $t_3 - t_2 \le 2BL$ possible queries in total. It follows that the expected encoding size in the second case is at most $\frac{3}{4} \cdot l\delta + L\lg(BL) + O(L)$, so the expected encoding size in general is at most $\frac{7}{8} \cdot l\delta + L\lg(BL) + O(L)$, which is the desired contradiction. (Recall that $l = \Theta(BL)$ and $B = \omega(1)$.) We have thus established the following main result:

THEOREM 5.1. *Any algorithm for the online partial-sums problem has an amortized cell-probe complexity of* $\Omega(1 + \lg n / \lg(b/\delta))$ *per operation.*

6 Conclusion

An interesting direction for future research would be to better understand the offline problem in the group model. It would also make sense to pose this problem in the RAM model, and we conjecture that the data structure presented in this paper is also optimal in the offline case. Unfortunately, however, our ability to prove lower bounds for offline problems on a RAM seems quite limited at present.

Most lower bound results, including the ones in this paper, are based on `sum` queries. However, it would also be interesting to prove a tight bound on the complexity of the `select` operation. The problem is especially interesting because it is a combination of two famous problems: the partial-sums and the predecessor problem. We found that a lower bound of $\Omega(\lg n / \lg b)$ is not hard to derive in our framework. In retrospect, this bound also stems from the work of [19]. The difficulty of improving this bound is that there seems to be no easy way to extract more than one bit of information from an answer to `select`.

Finally, we hope that the lower bound techniques developed in this paper will find further applications to interesting data-structure problems.

References

[1] S. Alstrup, M. A. Bender, E. D. Demaine, M. Farach-Colton, J. I. Munro, T. Rauhe, and M. Thorup, *Efficient Tree Layout in a Multilevel Memory Hierarchy*, arXiv:cs.DS/0211010, `http://www.arXiv.org/abs/cs.DS/0211010`.

[2] S. Alstrup, T. Husfeldt, and T. Rauhe, *Marked ancestor problems*, in Proc. 39th Annual Symposium on Foundations of Computer Science (FOCS'98), 534–543.

[3] A. Andersson, P. B. Miltersen, and M. Thorup, *Fusion Trees can be Implemented with AC^0 Instructions Only*, Theoretical Computer Science 215(1–2):337–344, 1999.

[4] P. Beame and F. E. Fich, *Optimal Bounds for the Predecessor Problem*, in Proc. 31st ACM Symposium on Theory of Computing (STOC'99), 295–304.

[5] A. M. Ben-Amram and Z. Galil, *A Generalization of a Lower Bound Technique due to Fredman and Saks*, Algorithmica 30(1):34–66, 2001.

[6] A. M. Ben-Amram and Z. Galil, *Lower Bounds for Dynamic Data Structures on Algebraic RAMs*, Algorithmica 32(3):364–395, 2002.

[7] M. A. Bender, G. S. Brodal, R. Fagerberg, D. Ge, S. He, H. Hu, J. Iacono, and A. López-Ortiz, *The Cost of Cache-Oblivious Searching*, in Proc. 44th Annual Symposium on Foundations of Computer Science (FOCS'03), to appear.

[8] W. A. Burkhard, M. L. Fredman, and D. J. Kleitman, *Inherent complexity trade-offs for range query problems*, Theoretical Computer Science 16:279–290, 1981.

[9] B. Chazelle, *Lower Bounds for Off-Line Range Searching*, Discrete & Computational Geometry 17(1):53–65, 1997.

[10] P. F. Dietz, *Optimal algorithms for list indexing and subset rank*, in Proc. Workshop on Algorithms and Data Structures (WADS'89), 39–46.

[11] P. M. Fenwick, *A new data structure for cumulative frequency tables*, Software: Practice and Experience 24(3):327–336, 1994.

[12] G. S. Frandsen, P. B. Miltersen, and S. Skyum, *Dynamic word problems*, Journal of the ACM 44(2):257–271, 1997.

[13] M. L. Fredman, *A lower bound on the complexity of orthogonal range queries*, Journal of the ACM 28:696–705, 1981.

[14] M. L. Fredman, *The complexity of maintaining an array and computing its partial sums*, Journal of the ACM 29:250–260, 1982.

[15] M. L. Fredman and M. E. Saks, *The cell probe complexity of dynamic data structures*, in Proc. 21st ACM Symposium on Theory of Computing (STOC'89), 345–354.

[16] M. L. Fredman and D. E. Willard, *Surpassing the information theoretic bound with fusion trees*, Journal of Computer and System Sciences 47:424–436, 1993.

[17] H. Hampapuram and M. L. Fredman, *Optimal Biweighted Binary Trees and the Complexity of Main-*

taining Partial Sums, SIAM Journal on Computing 28(1):1–9, 1998.

[18] W. K. Hon, K. Sadakane and W. K. Sung, *Succinct Data Structures for Searchable Partial Sums*, in Proc. 14th Annual International Symposium on Algorithms and Computation (ISAAC 2003), to appear.

[19] T. Husfeldt and T. Rauhe, *New Lower Bound Techniques for Dynamic Partial Sums and Related Problems*, SIAM Journal on Computing 32(3):736–753, 2003.

[20] T. Husfeldt, T. Rauhe and S. Skyum, *Lower Bounds for Dynamic Transitive Closure, Planar Point Location, and Parentheses Matching*, Nordic Journal of Computing 3(4):323–336, 1996.

[21] M. Li and P. Vitányi, *An Introduction to Kolmogorov Complexity and its Applications*, Springer-Verlag, New York, second edition, 1997.

[22] P. B. Miltersen, *Cell probe complexity - a survey*, Advances in Data Structures Workshop, Conference on the Foundations of Software Technology and Theoretical Computer Science (FSTTCS'99).

[23] R. Raman, V. Raman, and S. S. Rao, *Succinct Dynamic Data Structures*, in Proc. Workshop on Algorithms and Data Structures (WADS'01), 426–437.

[24] A. C. Yao, *On the complexity of maintaining partial sums*, SIAM Journal on Computing 14:277–288, 1985.