LINEAR TIME AND MEMORY-EFFICIENT COMPUTATION

KENNETH W. REGAN*

Abstract. A realistic model of computation called the *Block Move* (BM) model is developed. The BM regards computation as a sequence of finite transductions in memory, and operations are timed according to a memory cost parameter μ. Unlike previous memory-cost models, the BM provides a rich theory of linear time, and in contrast to what is known for Turing machines, the BM is proved to be highly *robust* for linear time. Under a wide range of μ parameters, many forms of the BM model, ranging from a fixed-wordsize RAM down to a single finite automaton iterating itself on a single tape, are shown to simulate each other up to constant factors in running time. The BM is proved to enjoy efficient universal simulation, and to have a tight deterministic time hierarchy. Relationships among BM and TM time complexity classes are studied.

Key Words. Computational complexity, theory of computation, machine models, Turing machines, random-access machines, simulation, memory hierarchies, finite automata, linear time, caching.

AMS(MOS) subject classification. 68Q05, 68Q15, 68Q25, 68Q68.

1. Introduction. This paper develops a new theory of linear-time computation. The *Block Move* (BM) model introduced here extends ideas and formalism from the *Block Transfer* (BT) model of Aggarwal, Chandra, and Snir [2]. The BT is a random access machine (RAM) with a special *block transfer* operation, together with a parameter $\mu : \mathbf{N} \rightarrow \mathbf{N}$ called a *memory access cost function*. The RAM's registers are indexed 0,1,2,..., and $\mu(a)$ denotes the cost of accessing register a. A block transfer has the form

$$\textit{copy } [a_1 \ldots b_1] \textit{ into } [a_2 \ldots b_2],$$

and is *valid* if these intervals have the same size m and do not overlap. With regard to a particular μ, the charge for the block transfer is $m + \mu(c)$ time units, where $c = \max\{ a_1, b_1, a_2, b_2 \}$. The idea is that after the initial charge of $\mu(a)$ for accessing the two blocks, a line of consecutive registers can be read or written at unit time per item. This is a reasonable reflection of how pipelining can hide memory latency, and accords with the behavior of physical memory devices (see [3], p1117, or [34], p 214). An earlier paper [1] studied a model called HMM which lacked the block-transfer construct. The main memory cost functions treated in these papers are $\mu_{\log}(a) := \lceil \log_2(a+1) \rceil$, which reflects the time required to write down the memory address a, and the functions $\mu_d(a) := \lceil a^{1/d} \rceil$ with $d = 1, 2, 3, \ldots$, which model the asymptotic increase in communication time for memory laid out on a d-dimensional grid. (The cited papers write f in place of μ and α for $1/d$.) The two-level *I/O complexity* model of Aggarwal and Vitter [3] has fixed block-size and a fixed cost for accessing the outer level, while the *Uniform Memory Hierarchy* (UMH) model of Alpern, Carter, and Feig [5] scales block-size and memory access cost upward in steps at higher levels.

The BM makes the following changes to the BT. First, the BM fixes the wordsize of the underlying machine, so that registers are essentially the same as cells on a Turing tape. Second, the BM provides native means of shuffling and reversing blocks.

* Department of Computer Science, State University of New York at Buffalo, 226 Bell Hall, Buffalo, NY 14620-2000. Supported in part by NSF Research Initiation Award CCR-9011248.

Third and most important, the BM allows other finite transductions S besides *copy* to be applied to the data in a block operation. A *block move* has the form

$$S\ [a_1 \ldots b_1]\ \textit{into}\ [a_2 \ldots b_2].$$

If x is the string formed by the symbols in cells a_1 through b_1, this means that $S(x)$ is written to the tape beginning at cell a_2 in the direction of b_2, with the proviso that a blank B appearing in the output $S(x)$ leaves the previous content of the target cell unchanged. This proviso implements *shuffle*, while *reverse* is handled by allowing $b_1 < a_1$ and/or $b_2 < a_2$. The block move is *valid* if the two intervals are disjoint, and meets the *strict boundary condition* if $S(x)$ neither overflows nor underflows $[a_2 \ldots b_2]$. The *work* performed in the block move is defined to be the number $|x|$ of bits read, while the *memory access charge* is again $\mu(c)$, $c = \max\{a_1, b_1, a_2, b_2\}$. The μ-*time* is the sum of these two numbers. Adopting terms from [5], we call a BM M *memory-efficient* if the total memory access charges stay within a constant factor (depending only on M) of the work performed, and *parsimonious* if the ratio of access charges to work approaches 0 as the input length n increases.

In the BT model, Aggarwal, Chandra, and Snir [2] proved tight nonlinear lower bounds of $\Theta[n \log n]$ with $\mu = \mu_1$, $\Theta[n \log\log n]$ with $\mu = \mu_d$, $d > 1$, and $\Theta[n \log^* n]$ with $\mu = \mu_{\log}$, for the so-called "Touch Problem" of executing a sequence of operations during which every value in registers $R_1 \ldots R_n$ is copied at least once to R_0. Since any access to R_a is charged the same as copying R_a to R_0, this gives lower bounds on the time for any BT computation that involves all of the input. In the BM model, however, the other finite transductions can glean information about the input in a way that *copy* cannot. Even under the highest cost function μ_1 that we consider, many interesting nonregular languages and functions are computable in linear time.

Previous models. It has long been realized that the standard unit-cost RAM model [21, 31, 18] is too powerful for many practical purposes. Feldman and Shapiro [22] contend that realistic models $\mathcal{M}$, both sequential and parallel, should have a property they call "polynomial vicinity" which we state as follows: Let C be a data configuration, and let H_C stand for the finite set of memory locations [or data items] designated as "scanned" in C. For all $t > 0$, let I_t denote the set of locations [or items] i such that there exists an $\mathcal{M}$-program that, when started in configuration C, scans i within t time units. Then the model $\mathcal{M}$ has *vicinity* $v(t)$ if for all C and t, $|I_t|/|H_C| \leq v(t)$. In 3D space, real machines "should have" at most cubic vicinity. The RAM model, however, has exponential vicinity even under the *log-cost criterion* advocated by Cook and Reckhow [18]. So do the random-access Turing machine (RAM-TM) forms described in [30, 26, 7, 14, 64], and TMs with tree-structured tapes (see [57, 63, 51, 52]). Turing machines with d-dimensional tapes (see [31, 60, 50]) have vicinity $O(t^d)$, regardless of the number of such tapes or number of heads on each tape, even with head-to-head jumps allowed. The standard TM model, with $d = 1$, has linear vicinity. The "RAM with polynomially compact memory" of Grandjean and Robson [29] limits integers i that can be stored and registers a that can be used to a polynomial in the running time T. This is not quite the same as polynomial vicinity—if $t \ll T$, the machine within t steps could still address a number of registers that is exponential in t. The BM has polynomial vicinity under μ_d (though not under $\mu_{\log}$), because any access outside the first t^d cells costs more than t time units. The theorem of [56] that deterministic linear time on the standard TM (DLIN) is properly contained in nondeterministic TM linear time (NLIN) is not known to carry over to any model of super-linear vicinity.

Practical motivations. The BM attempts to capture, with a minimum of added notation, several important properties of computations on real machines that the previous models neglect or treat too coarsely. The motivations are largely the same as those for the BT and UMH: As calibrated by μ, memory falls into a *hierarchy* ranging from relatively small amounts of low-indexed fast memory up through to large amounts of slow external storage. An algorithm that enjoys good *temporal locality of reference*, meaning that long stretches of its operation use relatively few different data items, can be implemented as a BM program that first copies the needed items to low memory (figuratively, to a cache), and is rewarded by a lower sum of memory-access charges. Good *spatial locality of reference*, meaning that needed data items are stored in neighboring locations in approximately the order of their need, is rewarded by the possibility of batching or pipelining a sequence of operations in the same block move. However, the BM appears to emphasize the sequencing of data items within a block more than the BT and UMH do, and we speak more specifically of *good serial access* rather than spatial locality of reference. The BM breaks sequential computation into phases in which access is serial and the operation is a finite transduction, and allows "random" access only between phases. Both μ-*time*(n) and the count $R(n)$ of block moves provide ways to quantify random access as a resource. The latter also serves as a measure of parallel time, since finite transductions can be computed by parallel prefix sum. Indeed, the BM is similar to the Pratt-Stockmeyer vector machine [61], and can also be regarded as a fixed-wordsize analogue of Blelloch's "scan" model [11].

Results. The first main theorem is that the BM is a very *robust* model. Many diverse forms of the machine simulate each other up to constant factors in μ-time, under a wide range of cost functions μ. Allowing multiple tapes or heads, expanding or limiting the means of tape access, allowing invalid block moves, making block boundaries pre-set or data-dependent in a block move, even reducing the model down to a single finite automaton that iterates itself on a single tape, makes no or little difference. We claim that this is the first sweeping *linear-time* robustness result for a natural model of computation. A "linear speed-up" theorem, similar to the familiar one for Turing machines, makes the constant factors on these simulations as small as desired. All of this gives the complexity measure μ-*time* a good degree of machine-independence. Some of the simulations preserve the work (w) and memory-access charges (μ-*acc*) separately, while others trade w off against μ-*acc* to preserve their sum.

Section 2 defines the basic BM model and also the reduced form. Section 3 defines all the richer forms, and Section 4 proves their equivalence. The linear speed-up theorem and some results on memory-efficiency are in Section 5. The second main result of this paper, in Section 6, shows that like the RAM but unlike what is known for the standard multitape Turing machine model (see [36, 24]), the BM carries only a constant factor overhead for universal simulation. The universal BM given is efficient under any μ_d, while separate constructions work for $\mu_{\log}$. In consequence, for any fixed $\mu = \mu_d$ or $\mu_{\log}$, the BM complexity classes DμTIME$[t]$ form a *tight deterministic time hierarchy* as the order of the time function t increases. Whether there is any hierarchy at all when μ rather than t varies is shown in Section 7 to tie back to older questions of determinism versus nondeterminism. This section also compares the BM to standard TM and RAM models, and studies BM complexity classes. Section 8 describes open problems, and Section 9 presents conclusions.

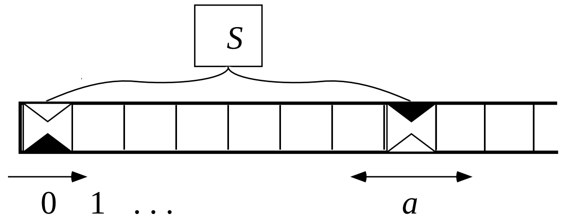

FIG. 1. *BM with allowed head motions in a pass*

2. The Block Move Model. We use λ for the empty string and B for the *blank* character. $\mathbf{N}$ stands for $\{0, 1, 2, 3, \ldots\}$. Characters in a string x of length m are numbered $x_0 x_1 \cdots x_{m-1}$. We modify the *generalized sequential machine* (GSM) of [36] so that it can exit without reading all of its input.

DEFINITION 2.1. A *generalized sequential transducer* (GST) is a machine S with components $(Q, \Gamma, \delta, \rho, s, F)$, where $F \subseteq Q$ is the set of *terminal states*, $s \in Q \setminus F$ is the *start state*, $\delta : (Q \setminus F) \times \Gamma \to Q$ is the *transition function*, and $\rho : (Q \setminus F) \times \Gamma \to \Gamma^*$ is the *output function*. The *I/O alphabet* Γ may contain the blank B.

A sequence $(q_0, x_0, q_1, x_1, \ldots, q_{m-1}, x_{m-1}, q_m)$ is a *halting trajectory* of S on input x if $q_0 = s$, $q_m \in F$, $x_0 x_1 \ldots x_{m-1}$ is an initial substring of x, and for $0 \leq i \leq m-1$, $\delta(q_i, x_i) = q_{i+1}$. The *output* $S(x)$ is then defined to be $\rho(q_0, x_0) \cdot \rho(q_1, x_1) \cdots \rho(q_{m-1}, x_{m-1})$.

By common abuse of notation we also write $S(\cdot)$ for the partial function computed by S. Except briefly in Section 8, all finite state machines we consider are deterministic. A symbol c is an *endmarker* for a GST S if every transition on c sends S to a terminal state. Without loss of generality, B is an endmarker for all GSTs.

The intuitive picture of our model is a "circuit board" with GST "chips," each of which can process streams of data drawn from a single tape. The formalism is fairly close to that for Turing machines in [36].

DEFINITION 2.2. A *Block Machine* (BM) is denoted by $M = (\mathcal{Q}, \Sigma, \Gamma, \delta, B, S_0, F)$, where:

- $\mathcal{Q}$ is a finite set consisting of *GSTs*, *move states*, and *halt states*.
- F is the set of halt states.
- Every GST has one of the four labels Ra, La, $0R$, or $0L$.
- *Move states* are labeled either $\lfloor a/2 \rfloor$, $2a$, or $2a+1$.
- Σ is the I/O alphabet of M, while the work alphabet Γ is used by all GSTs.
- The *start state* S_0 is a GST with label Ra.
- The *transition function* δ is a mapping from $(\mathcal{Q} \setminus F) \times \Gamma$ to $\mathcal{Q}$.

We find it useful to regard GSTs as "states" in a BM machine diagram, reading the machine in terms of the specific functions they perform, and submerging the individual states of the GSTs onto a lower level. M has two tape heads, called the "cell-0 head" and the "cell-a head," which work as follows in a GST pass (Figure 1). Let $\sigma[i]$ stand for the symbol in tape cell i, and for $i, j \in \mathbf{N}$ with $j < i$ allowed, let $\sigma[i \ldots j]$ denote the string formed by the symbols from cell i to cell j.

DEFINITION 2.3. A *pass* by a GST S in a BM works as follows, with reference to the *current address* a and each of the four *modes* $Ra, La, 0R, 0L$:

(Ra) S reads the tape moving rightward from cell a. Since B is an endmarker for S, there is a cell $b \geq a$ in which S exits. Let $x := \sigma[a \ldots b]$ and $y := S(x)$. If $y = \lambda$, the pass ends with no change in the tape. For $y \neq \lambda$, let $c := |y| - 1$. Then y is written into cells $[0 \ldots c]$, except that if $y_i = B$, cell i is left unchanged. This completes the pass.

(La) S reads the tape moving leftward from cell a. Unless S runs off the left end of the tape (causing a "crash"), let $b \leq a$ be the cell in which S exits. As before let $x := \sigma[a \ldots b]$, $y := S(x)$, and if $y \neq \lambda$, $c := |y| - 1$. Then formally, for $0 \leq i \leq c$, if $y_i \neq B$ then $\sigma[i] := y_i$, while if $y_i = B$ then $\sigma[i]$ is unchanged.

($0R$) S reads from cell 0, necessarily moving right. Let c be the cell in which S halts. Let $x := \sigma[0 \ldots c]$, $y := S(x)$, and $b := a + |y| - 1$. Then y is written rightward from a into cells $[a \ldots b]$, with the same convention about B as above.

($0L$) Same as $0R$, except that $b := a - |y| + 1$, and y is written leftward from a into $[a \ldots b]$.

Here a, b, and c are the *access points* of the pass. Each of the four kinds of pass is *valid* if either (i) $y = \lambda$, (ii) $a, b, c \leq 1$, or (iii) $c < \min\{a, b\}$. The case $y = \lambda$ is called an *empty pass*, while if $|x| = 1$, then it is called a *unit pass*.

In terms of Section 1, Ra and La execute the block move $S\,[a \ldots b]$ *into* $[0 \ldots c]$, except that the boundaries b and c are not set in advance and can depend on the data x. Similarly $0R$ and $0L$ execute $S\,[0 \ldots c]$ *into* $[a \ldots b]$. We make the distinction is that in a *pass* the read and write boundaries may depend on the data, while in a *block move* (formalized in the next section) they are set beforehand. The tape is regarded as linear for passes or block moves, but as a binary tree for addressing. The root of the tree is cell 1, while cell 0 is an extra cell above the root. The validity condition says that the intervals $[a \ldots b]$ and $[0 \ldots c]$ must not overlap, with a technically convenient exception in case the whole pass is done in cells 0 and 1. If a pass is invalid, M is considered to "crash." A pass of type Ra or La figuratively "pulls" data to the left end of the tape, and we refer to it as a *pull*; similarly we call a pass of type $0R$ or $0L$ a *put*. Furthering the analogy to internal memory or to a processor cache, these pass types might be called a *fetch* and *writeback*, respectively. An La or $0L$ pass can reverse a string on the tape.

Definition 2.4. A *valid computation* $\vec{c}$ by a BM $M = (Q, \Sigma, \Gamma, \delta, B, S_0, F)$ is defined as follows. Initially $a = 0$, the tape contains x in cells $0 \ldots |x| - 1$ with all other cells blank, and S_0 makes the first pass. When a pass by a GST S ends, let c be the character read on the transition in which S exited. Then control passes to $\delta(S, c)$. In a move state q, the new current address a' equals $\lfloor a/2 \rfloor$, $2a$, or $2a+1$ according to the label of q, and letting d be the character in cell a', control passes to state $\delta(q, d)$. All passes must be valid, and a valid computation ends when control passes to a halting state. Then the *output*, denoted by $M(x)$, is defined to be $\sigma[0 \ldots m-1]$, where $\sigma[m]$ is the leftmost non-Σ character on the tape. If M is regarded as an acceptor, then the language of strings accepted by M is denoted by $L(M) := \{\, x \in \Sigma^* | M(x) \text{ halts and outputs } 1 \,\}$.

The convention on output is needed since a BM cannot erase, i.e. write B. Alternatively, for an acceptor, F could be partitioned into states labeled Accept and Reject.

Definition 2.5. A *memory cost function* is any function $\mu : \mathbf{N} \to \mathbf{N}$ with the properties (a) $\mu(0) = 0$, (b) $(\forall a)\mu(a) \leq a$, and (c) $(\forall N \geq 1)(\forall a)\mu(Na) \leq N\mu(a)$.

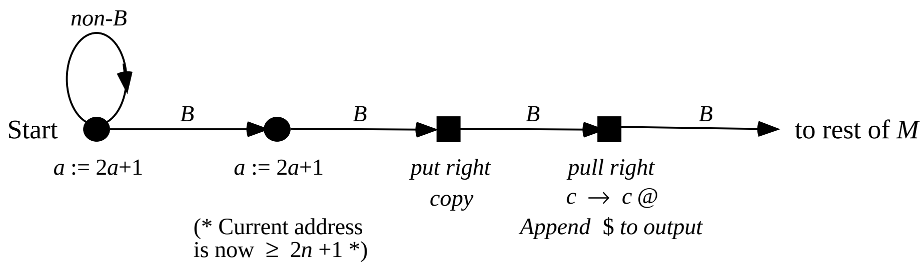

FIG. 2. *A BM that makes a fresh track*

Our results will only require the property (c′): $(\forall N \geq 1)(\exists N' \geq 1)\ (\forall^\infty a)\ \mu(Na) \leq N'\mu(a)$. While property (c) can be named by saying that μ is "sub-linear," we do not know a standard mathematical name for (c′), and we prefer to call either (c) or (c′) the *tracking property* for the following reason:

EXAMPLE 2.1. *Tracking.* Figure 2 diagrams a multichip BM routine that changes the input $x = x_0x_1\cdots x_{n-1}$ to $x_0@x_1@\cdots x_{n-2}@x_{n-1}@\$$, where @ acts as a "surrogate blank," and only @ or B appears to the right of the \$. This divides the tape into two *tracks* of odd and even cells. A BM can write a string y to the second track by pulling it as $By_0By_1\cdots By_{m-1}By_m$, since the blanks B leave the contents of the first track undisturbed. Two strings can also be *shuffled* this way. Since $\mu(2a) \leq 2\mu(a)$, the tracking no more than doubles the memory access charges.

The principal memory cost functions we consider in this paper are the *log-cost function* $\mu_{\log}(a) := \lceil \log_2(a+1) \rceil$, and for all $d \geq 1$, the *d-dimensional layout function* $\mu_d(a) := \lceil a^{1/d} \rceil$. These have the tracking property.

DEFINITION 2.6. For any memory cost function μ, the *μ-time* of a valid pass that reads x and operates the cell-a head in the interval $[a \ldots b]$ is given by $\mu(a)+|x|+\mu(b)$. The *work* of the pass is $|x|$, and the *memory access charge* is $\mu(a) + \mu(b)$. A move state that changes a to a' performs 1 unit of work and has a memory access charge of $\mu(a) + \mu(a')$. The sum of the work over all passes in a valid computation $\vec{c}$ is denoted by $w(\vec{c})$, the total memory access charges by μ-$acc(\vec{c})$, and the total *μ-time* by $\mu(\vec{c}) := w(\vec{c}) + \mu\text{-}acc(\vec{c})$.

Intuitively, the charge for a pass is $\mu(a)$ time units to access cell a, plus $|x|$ time units for reading or writing the block, plus $\mu(b)$ to communicate to the CPU that the pass has ended and to re-set the heads. We did not write $\max\{\,\mu(a), \mu(b)\,\}$ because b is not known until after the time to access a has already been spent; this makes no difference up to a factor of two. Replacing $|x|$ by $|x| + |S(x)|$ or by $\max\{\,|x|, |S(x)|\,\}$, or adding $\mu(c)$ to $\mu(a) + \mu(b)$, also make no difference in defining w or μ-acc, this time up to a constant factor that may depend on M.

DEFINITION 2.7. For any input x on which a BM M has halting computation $\vec{c}$, we define the complexity measures

Work:	$w(M,x) := w(\vec{c})$.
Memory access:	$\mu\text{-}acc(M,x) := \mu\text{-}acc(\vec{c})$.
μ-time:	$\mu\text{-}time(M,x) := w(M,x) + \mu\text{-}acc(M,x)$.
Space:	$s(M,x) :=$ the maximum of a for all access points a in $\vec{c}$.
Pass count:	$R(M,x) :=$ the total number of passes in $\vec{c}$.

M is dropped when it is understood, and the above are extended in the usual manner to functions $w(n)$, μ-$acc(n)$, μ-$time(n)$, $s(n)$, and $R(n)$ by taking the maximum over all inputs x of length n. A measure of space closer to the standard TM space measure could be defined in the extended BM models of the next section by placing the input x on a separate read-only input tape, but we do not pursue space complexity further in this paper. The pass count appears to be sandwiched between two measures of *reversals* for multitape Turing machines, namely the now-standard one of [59, 35, 16], and the stricter notion of [43] which essentially counts keeping a TM head stationary as a reversal.

DEFINITION 2.8. For any memory cost function μ and recursive function $t : \mathbf{N} \to \mathbf{N}$, D$\mu$TIME$[t]$ stands for the class of languages accepted by BMs M that run in time $t(n)$, i.e. such that for all x, $\mu\text{-}time(M, x) \leq t(|x|)$. TLIN stands for D$\mu_1$TIME$[O(n)]$.

We also write DμTIME$[t]$ and TLIN for the corresponding function classes. Section 7 shows that TLIN is contained in the TM linear-time class DLIN. We argue that languages and functions in TLIN have true linear-time behavior even under the most constrained implementations.

We do not separate out the work performed from the total memory access charges in defining BM complexity classes, but do so in adapting the following notions and terms from [5] to the BM model.

DEFINITION 2.9. (a) A BM M is *memory efficient*, under a given memory cost function μ, if there is a constant K such that for all x, $\mu\text{-}time(M, x) \leq K \cdot w(M, x)$.
(b) M is *parsimonious* under μ if $\mu\text{-}time(M, x)/w(M, x) \to 1$ as $|x| \to \infty$.

Equivalently, M is memory efficient under μ if $\mu\text{-}acc(M, x) = O(w)$, and parsimonious under μ if $\mu\text{-}acc(M, x) = o(w)$, where the asymptotics are as $|x| \to \infty$. The intuition, also expressed in [5], is that efficient or parsimonious programs make good use of a memory cache.

Definition 2.9 does not imply that the given BM M is optimal for the function f it computes. Indeed, from *Blum's speed-up theorem* [12] and the fact that μ-*time* is a complexity measure, there exist computable functions with no μ-*time* optimal programs at all. To apply the concepts of memory efficiency and parsimony to languages and functions, we use the following relative criterion:

DEFINITION 2.10. (a) A function f is *inherently μ-efficient* if for every BM M_0 that computes f, there is a BM M_1 which computes f and a constant $K > 0$ such that for all x, $\mu\text{-}time(M_1, x) \leq K \cdot w(M_0, x)$.
(b) f is *inherently μ-parsimonious* if for every BM M_0 computing f there is a BM M_1 computing f such that $\limsup_{|x| \to \infty} \mu\text{-}time(M_1, x)/w(M_0, x) \leq 1$.

By definition μ-parsimony $\Longrightarrow$ μ-efficiency, and if f is inherently efficient (resp. parsimonious) under μ_1, then f is inherently efficient (resp. parsimonious) under every memory cost function $\mu \leq \mu_1$.

Just for the next three examples, we drop the validity condition on rightward pulls; that is, we allow the tape intervals $[a \dots b]$ and $[0 \dots c]$ to overlap in an Ra move. This is intuitively reasonable so long as the cell-0 head does not overtake the cell-a head and write over a cell that the latter hasn't read yet. Theorem 4.1 will allow us to drop the validity condition with impunity, but the proof of Theorem 2.1 below requires that it be in force.

EXAMPLE 2.2. *Balanced Parentheses.* Let D_1 stand for the language of balanced parenthesis strings over $\Sigma := \{\, (,) \,\}$. Let the GST S work as follows on any $x \in \Sigma^*$:

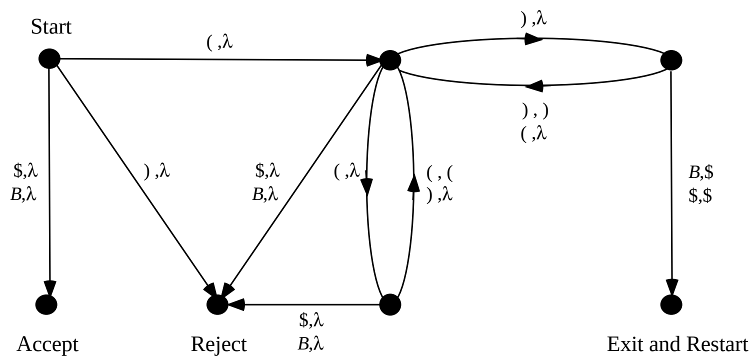

FIG. 3. *Reduced-form BM for the language of balanced parentheses*

If $x = \lambda$, S goes to a terminal state marked ACCEPT; if x begins with ')', S goes to REJECT. Else S erases the leading '(' and thereafter takes bits in twos, translating

$$((\mapsto (\qquad)) \mapsto) \qquad () \mapsto \lambda \qquad)(\mapsto \lambda. \tag{1}$$

If x ends in '(' or $|x|$ is odd, S also signals REJECT. Then S has the property that for any $x \neq \lambda$ that it doesn't immediately reject, $x \in D_1 \iff S(x) \in D_1$. Furthermore, $|S(x)| < |x|/2$. We can think of D_1 as being self-reducible in a particularly sharp sense.

Figure 3 shows the corresponding BM in the "reduced form" defined below. The '\$' endmarker is written on the first pass, and prevents leftover "garbage" on the tape from interfering with later passes. We take this for granted in some later descriptions of BMs. For any memory cost function μ, the running time of M is bounded by

$$\sum_{i=0}^{log_2 n} \mu(0) + 2^i + \mu(2^i), \tag{2}$$

which is $O(n)$ even for $\mu = \mu_1$. Hence the language D_1 belongs to TLIN.

EXAMPLE 2.3. *Counting.* Let $\Sigma := \{\mathrm{a}, \mathrm{b}\}$. We can build a GST S with alphabet $\Gamma = \{\mathrm{a}, \mathrm{b}, 0, 1, \$, B\}$ that runs as follows on inputs of the form $x' = xu\$$ with $x \in \{\mathrm{a}, \mathrm{b}\}^*$ and $u \in \{0, 1\}^*$: S erases bits $x_0, x_2, x_4, \ldots$ of x and remembers $|x|$ modulo 2. S then copies u, and on reading the final \$ (or on the first pass, B), S outputs 0\$ if $|x|$ was even, 1\$ if $|x|$ was odd. S is also coded so that if $x = \lambda$, S goes to HALT. Let M be the BM which iterates S on input x. Then $M(x)$ halts with $|x|$ in binary notation on its tape (followed by '\$' and "garbage"). The μ-time for this iteration is likewise $O(n)$ even for $\mu = \mu_1$.

EXAMPLE 2.4. *Simulating a TM.* Let $T := (Q, \Sigma, \Gamma, \delta, B, q_0, F)$ be a single-tape TM in the notation of [36]. Define the *ID alphabet* of T to be $\Gamma_I := (Q \times \Gamma) \cup \Gamma \cup \{\wedge, \$\}$, where $\wedge, \$ \notin \Gamma$. The simulating BM M on an input $x = x_0 \cdots x_{n-1}$ makes a rightward pull that lays down the delimited initial ID $\wedge(q_0, x_0)x_1x_2 \cdots x_{n-1}\$$ of $T(x)$. The finite control of T is turned into a single GST S with alphabet Γ_I that produces successive IDs in the computation with each pass. Whenever T writes a blank, M writes @. Let T be programmed to move its head to cell 0 before halting. Then the final pass by M removes the $\wedge$ and \$ and leaves exactly the output $y := T(x)$ on the tape. Actually, because a BM cannot erase tape cells, y would be followed by some number of symbols @, but Definition 2.4 still makes y the output of M. Hence the BM is a universal model of computation.

The machines in Examples 2.2–2.4 only make rightward pulls from cell 0. Each is really a GST that iterates on its own output, a form generally known as a "cascading finite automaton" (CFA). Up to small technical differences, CFAs are comparable to the one-way "sweeping automata" studied by Ibarra et.al. [39, 41, 40, 37, 38, 15]. These papers characterize both one-way and two-way arrays of identical finite-state machines in terms of these and other automata and language classes. The following shows that the BM can be regarded as a generalization of these arrays, insofar as a BM can dynamically change its origin point a and direction of operation.

DEFINITION 2.11. The *reduced form* of the BM model consists of a single GST S whose terminal states q have labels $l_1(q) \in \{ a, \lfloor a/2 \rfloor, 2a, 2a+1, \textsc{Halt} \}$ and $l_2(q) \in \{ Ra, La, 0R, 0L \}$. The initial pass has mode Ra with $a = 0$. Whenever a pass by S exits in some state q with $l_1(q) \neq \textsc{Halt}$, the labels $l_1(q)$ and $l_2(q)$ determine the address and mode for the next pass. Computations and complexity measures are defined as before.

THEOREM 2.1. *Every BM M is equivalent to a BM M' in reduced form, up to constant factors in all five measures of Definition 2.7.*

Proof. The idea is to combine all the GSTs of M into a single GST S and save the current state of M in cells 0 and 1. Each pass of M is simulated by at most six passes of M', except for a "staircase" of $O(\log n)$ moves at the end which is amortized into the constant factors. This simulation expands the alphabet but does not make any new tracks. The details are somewhat delicate, owing to the lack of internal memory when a pass by M' ends, and require the validity condition on passes. The full proof is in the Appendix. □

In both directions, the tape cells used by M and M' are almost exactly the same; i.e., M is simulated "in place." Hence we consider the BM and the reduced form to be essentially identical. The idea of gathering all GSTs into one works with even less technical difficulty for the extended models in the next section.

3. Extensions of the BM.

We consider five natural ways of varying the BM model: (1) Remove or circumvent the validity restriction on passes. (2) Provide "random addressing" rather than "tree access" in move states. (3) Provide delimiters a_1, b_1, a_2, b_2 for block moves $S\,[a_1 \dots b_1]$ *into* $[a_2 \dots b_2]$, where the cell b_1 in which S exits is determined or calculated in advance. (4) Require that for every such block move, b_2 is such that $S(x)$ exactly fills $[a_2 \dots b_2]$. (5) Provide multiple main tapes and GSTs that can read from and write to k-many tapes at once. These extensions can be combined. We define them in greater detail, and in the next section, prove equivalences among them and the basic model.

DEFINITION 3.1. A BM *with buffer mechanism* has a new tape called the *buffer tape*, and GST chips S with the following six labels and functions:

(RaB) The GST S reads x from the main tape beginning in cell a and writes $S(x)$ to the buffer tape. The output $S(x)$ must have no blanks in it, and it completely replaces any previous content of the buffer. Taking b to be the cell in which S exits, the μ-time is $\mu(a) + |x| + \mu(b)$ as before.

(LaB) As for RaB, but reading leftward from cell a.

(BaR) Here S draws its input x from the buffer, and $S(x)$ is written on the main tape starting in cell a. Blanks in $S(x)$ are allowed and treated as before. When S exits, even if it has not read all of the buffer tape, the buffer is flushed. With b the destination of the last output bit (or $b = a$ if none), the μ-time is likewise $\mu(a) + |x| + \mu(b)$.

(BaL) As for BaR, but writing $S(x)$ leftward from cell a.

($0B$) As for RaB, but using the cell-0 head to read the input, and μ-time $|x|+\mu(c)$.

($B0$) As for BaR, but using the cell-0 head to write the output; likewise μ-time $|x| + \mu(c)$.

All six types of pass are automatically valid. Further details of computations and complexity measures are the same as before. A BM *with limited buffer mechanism* has no GSTs with labels $B0$ or $0B$, and consequently has no cell-0 head.

The original BM's moves of type Ra or La can now be simulated directly by RaB or LaB followed by $B0$, while $0R$ or $0L$ is simulated by $0B$ followed by BaR or BaL. For the limited buffer mechanism the simulation is trickier, but for $\mu = \mu_d$ we will show that it can be done efficiently. The next extension allows "random access."

DEFINITION 3.2. The *address mechanism* adds an *address tape* and new *load* moves labeled RaA, LaA, and $0A$. These behave and are timed like the buffer moves RaB, LaB, and $0B$ respectively, but direct their output to the address tape instead. As with the buffer, the output completely replaces the previous content of the address tape. Addresses are written in binary notation with the least significant bit leftmost on the tape. The output a' of a load becomes the new current address. Move states may be discarded without loss of generality.

EXAMPLE 3.1. *Palindromes.* Let *Pal* denote the language of palindromes over a given alphabet Σ. We sketch a BM M with address mechanism that accepts *Pal*. On input x, M makes a fresh track on its tape via Example 2.1, and runs the procedure of Example 2.3 to leave $n := |x|$ in binary notation on this track. In running this procedure, we either exempt rightward pulls from the validity condition or give M the buffer mechanism as well. The fresh-track cell which divides the right half of x from the left half has address $n' := 2\lfloor n/2 \rfloor + 1$. A single $0A$ move can read n but copy the first bit as 1 to load the address n'. M then pokes a \$ into cell n'. Another load prepends a '0' so as to address cell $2n$, and M then executes a leftward pull that interleaves the left half of x with the right half. A bit-by-bit compare from cell 0 finishes the job. M also runs in linear μ_1-time.

The address mechanism provides for indirect addressing via a succession of loads, and makes it easy to implement pointers, linked lists, trees, and other data structures and common features of memory management on a BM, subject to charges for the number and size of the references.

Thus far, all models have allowed data-dependent block boundaries. We call any of the above kinds of BM M *self-delimiting* if there is a sub-alphabet Γ_e of *endmarkers* such that all GSTs in M terminate precisely on reading an endmarker. (If we weaken this property slightly to allow a GST S to exit on a non-endmarker on its second transition, then it is preserved in the proof of Theorem 2.1.) The remaining extensions pre-set the read block $[a_1 \dots b_1]$ and the write block $[a_2 \dots b_2]$, and this is when we speak of a *block move* rather than a *pass*. Having b_1 fixed would let us use the original GSM model from [36]. However, the machines that follow are always able to drop an endmarker into cell b_1 and force a GST S to read all of $[a_1 \dots b_1]$. Hence we may ignore the distinction and retain 'GST' for consistency.

DEFINITION 3.3. A *block move* is denoted by $S\,[a_1 \dots b_1]$ *into* $[a_2 \dots b_2]$ and has this effect on the tape: Let $x := \sigma[a_1 \dots b_1]$. Then $S(x)$ is written to the tape beginning at a_2 and proceeding in the direction of b_2, with the proviso that each blank in $S(x)$ leaves the target cell unchanged, as in Definition 2.3. The block move is *valid* so long as the intervals $[a_1 \dots b_1]$ and $[a_2 \dots b_2]$ are disjoint. It *underflows* if $|S(x)| < |b_2 - a_2| + 1$, and *overflows* if $|S(x)| > |b_2 - a_2| + 1$.

By default we tolerate underflows and overflows in block moves. We draw an analogy between the next form of the BM and a text editor in which the user may mark a source and destination block and perform an operation on them. One important point is that the BM does not allow insertions and deletions of the familiar "cut-and-paste" kind; instead, the output flows over the destination block and overwrites or lets stand according to the use of B in Definition 2.3. Willard [69] describes a model of a file system that lacks insertion and deletion, and gives fairly efficient algorithms for simulating them. Many text processors allow the user to define and move *markers* for points of immediate access in a file. Usually the maximum number of markers allowed is fixed to some number m. Adopting a term from data structures, we give the machine four *fingers*, with labels a_1, b_1, a_2, b_2, which can be assigned among the m markers and which delimit the source and destination blocks in any block move. Finger a_1 may be thought of as the "cursor." The dual use of "a_1" as the fixed label of a finger and as the number of the cell its assigned marker currently occupies may cause some confusion, but we try to keep the meanings clear below. The same applies to a_2, b_1, and b_2, and to later usage of these labels to name four special "address tapes."

DEFINITION 3.4. A *finger BM* has four *fingers* labeled a_1, b_1, a_2, b_2, and some number $m \geq 4$ of *markers*. Initially one marker is on the last bit of the input, while all other markers and all four fingers are on the first bit in cell 0. An invocation of a GST S executes the block move $S\,[a_1 \dots b_1]$ *into* $[a_2 \dots b_2]$. The *work* performed by the block move is $|b_1 - a_1| + 1$, while the *memory-access charge* is $\mu(c)$, where $c = \max\{\, a_1, b_1, a_2, b_2 \,\}$. In a move state, *each* marker on some cell a may be moved to cell $\lfloor a/2 \rfloor$, $2a$, or $2a{+}1$ (or kept where it is), and the four fingers may be redistributed arbitrarily among the markers. The cost of a move state is the maximum of $\mu(a)$ over all addresses a involved in finger or marker *movements*; those remaining stationary are not charged.

One classical difference between "fingers" and "pointers" is that there is no fixed limit on the number of pointers a program can create. Rather than define a form of the BM analogous to the *pointer machines* of Schönhage and others [45, 66, 67, 49, 10], we move straight to a model that uses "random-access addressing," a mechanism usually considered stronger than pointers (for in-depth comparisons, see [9, 10] and also [68]). The following BM form is based on a random-access Turing machine (RAM-TM; cf. "RTM" in [30] and "indexing TM" in [14, 64, 8]), and is closest to the BT.

DEFINITION 3.5. A *RAM-BM* has one *main tape*, four *address tapes* labeled a_1, b_1, a_2, b_2 and given their own heads, and a finite control comprised of *RAM-TM states* and *GST states*. In a RAM-TM state, the current main-tape address a is given by the content of tape a_1. The machine may read and change both the character in cell a and those scanned on the address tapes, and move each address tape head one cell left or right. In a GST state S, the address tapes give the block boundaries for the block move $S\ [a_1 \dots b_1]$ *into* $[a_2 \dots b_2]$ as described above, and control passes to some RAM-TM state. A RAM-TM step performs work 1 and incurs a memory-access charge of $\max\{\,\mu(a), \mu(b)\,\}$, where b is the rightmost extent of an address tape head. Block moves are timed as above. Both a RAM-TM step and a block move add 1 to the pass count $R(n)$. Other details of computations are the same as for the basic BM model.

A fixed-wordsize analogue of the original BT model of [2] can now be had by making *copy* the only GST allowed in block moves. A RAM-BM *with address loading* can use block moves rather than RAM-TM steps to write addresses.

DEFINITION 3.6. A finger BM or a RAM-BM obeys the *strict boundary condition* if in every block move $S\,[a_1 \ldots b_1]$ *into* $[a_2 \ldots b_2]$, $|S(x)|$ equals $|b_2 - a_2| + 1$.

This constraint is notable when S is such that $|S(x)|$ varies widely for different x of the same length. The next is a catch-all for further extensions.

DEFINITION 3.7. For $k \geq 2$, a *k-input GST* has k-many input tapes and one output tape, with $\delta : (Q \setminus F) \times \Gamma^k \rightarrow Q$ and $\rho : (Q \setminus F) \times \Gamma^k \rightarrow \Gamma^*$. Each input head advances one cell at each step.

DEFINITION 3.8. A *multitape BM* has some number $k \geq 2$ of main tapes, each possibly equipped with its own address and/or buffer tapes, and uses k-input GSTs in passes or block moves.

Further details of computations and complexity measures for multitape BMs can be inferred from foregoing definitions, and various validity and boundary conditions can be formulated. The proofs in the next section will make the workings of these machines clear.

Finally, given two machines M and M' of any kind and a cost function μ, we say *M' simulates M linearly in μ* if $\mu\text{-}time(M', x) = O(\mu\text{-}time(M, x)) + O(|x|)$. The extra '$O(n)$' is stated because like the RAM-TM, several BM variants give a sensible notion of computing in *sub*-linear time, while all the simulations to come involve an $O(n)$-time preprocessing phase to set up tracks on the main tape. Now we can state:

MAIN ROBUSTNESS THEOREM 3.1. *For any rational $d \geq 1$, all forms of the BM defined above simulate each other linearly in μ_d-time.*

If we adapted a standard convention for Turing machines to state that every BM on a given input x takes time at least $|x| + 1$ (cf. [36]), then we could say that all the simulations have constant-factor overheads in μ_d-time.

4. Proof of the Main Robustness Theorem. The main problems solved in the proof are: (1) how to avoid overlaps in reading and writing by "tape-folding" (Theorem 4.1), (2) how to simulate random access with one read head whose movements are limited (Lemma 4.6), and (3) how to precompute block boundaries without losing efficiency (Lemma 4.11 through Theorem 4.15). Analogues of these problems are known in other areas of computation, but solving them with only a constant factor overhead in μ-time requires some care. Some of the simulations give constant factor overheads in both w and μ-*acc*, but others trade off the work against the memory access charges. We also state bounds on w' and μ-*acc*$'$ for the simulating machine M' individually, and on the number R' of passes M' requires, in or after proofs. The space $s'(n)$ is always $O(s(n))$.

4.1. Simulations for data-dependent block boundaries. The first simulation uses the tracking property $\mu(Na) \leq N\mu(a)$ from Definition 2.5, and does not give constant-factor overheads in all measures. We give full details in this proof, in order to take reasonable shortcuts later.

THEOREM 4.1. *For every BM M with buffer there is a BM M' such that for every μ, M' simulates M linearly in μ-time.*

Proof. Let M have the buffer mechanism. Let C be the largest number of symbols output in any transition of any GST in M. Let $K := \lceil \log_2(2C+6) \rceil$ and $N := 2^K$. The BM M' first makes N-many tracks by iterating the procedure of Example 2.1. The track comprising cells $0, N, 2N, 3N, \ldots$ represents the main tape of M, while the two tracks flanking it are "marker tracks." The track through cells $2, N+2, \ldots$ represents the buffer tape. The other tracks are an "extension track," a "holding track," C-many "pull bays," and C-many "put bays." M' uses the symbol @ to reserve free space in tracks, and uses $\wedge$ and \$ to mark places in the tape. A \$ also delimits the buffer track so that leftover "garbage" does not interfere. Two invariants are that before every simulated pass by M with current address a, the current address a' of M' equals Na, and the tracks apart from the main and buffer tracks contain only blanks and @ symbols.

The move $a := 2a$ by M is simulated directly by $a' := 2a'$ in M'. The move $a := 2a+1$ is simulated by effecting $a' := \lfloor a'/2 \rfloor$ K-many times, then $a' := 2a'+1$, and then $a' := 2a'$ K-many times. The move $a := \lfloor a/2 \rfloor$ is simulated by effecting $a' := \lfloor a'/2 \rfloor$ $(K+1)$-many times, and then $a' := 2a$ K-many times. Since K is a constant, the overhead in μ-*acc* for each move is constant. Henceforth we refer to "cell a on the main track" in place of a'.

We need only describe how M' simulates each of the six kinds of pass by M. Since M has the $0B$ and $B0$ instructions, we may assume that the current address a for the other four kinds is always ≥ 1. For each state q of a GST S of M, M' has a GST S'_q which simulates S starting in state q, and which exits only on the endmarker \$. We write just S' when $q = s$ or q is understood.

(a) *RaB.* M' chooses $a_1 := 2a$, pokes $\wedge$ to the left of cell a, and pokes \$ to the left of cell a_1. M' then pulls $y_1 := S'[a \ldots a_1 - 1]$ to the C-many pull bays. By the choice of C, $|y_1| \leq Ca$, and so the pull is valid.

If the cell b in which S exits falls in the interval $[a \ldots a_1 - 1]$, then S' likewise exits in cell b. Since the exit character has no \$, the transition out of S' communicates that S has exited. M' then makes $(K+1)$-many moves $a := 2a$ so that M' now addresses cell Na_1 on the main track, which is cell $N^2 a_1$ overall. M' puts $y := y_1$ onto the extension track and then pulls y onto the buffer track. One more put then overwrites the used portion of the extension track with @ symbols. M' then effects $a := \lfloor a/2 \rfloor$ $(K+1)$-many times so that it addresses the original cell a again, and re-simulates S in order to overwrite the copy of y on the pull bays by @ symbols. All of these passes are valid. M' finally removes the $\wedge$ and \$ markers at cells a and a_1. The original time charge to M was $\mu(a) + m + (b)$, where $m = b - a + 1$. The time charged to M' in this case is bounded by:

$$
\begin{aligned}
\mu(Na) \;& + 2 + \mu(Na-1) + \mu(Na_1) + 2 + \mu(Na_1 - 1) && \text{(poke } \wedge \text{ and \$)} \\
& + \mu(Na) + Nm + \mu(Nb) && \text{(simulate } S\text{)} \\
& + 2K\mu(N^2 a_1) && \text{(move to cell } Na_1\text{)} \\
& + 3\mu(N^2 a_1) + 3N^2 m + 3\mu(N^2 a_1 + N^2(m-1)) && \text{(put and pull } y\text{)}
\end{aligned}
$$

$$+ 2K\mu(N^2 a_1) + \mu(Na) + Nm + \mu(Nb) + 2\mu(Na) + 4 + 2\mu(Na_1) \quad \text{(clean up)}$$

$$\leq (14N + 8N^2K + 12N^2)\mu(a) + (3N^2 + 2N)m + 2N\mu(b) + 4. \qquad (m-1 \leq a)$$

So far both the work w' and the memory access charges μ-acc' to M' are within a constant factor of the corresponding charges to M.

If S does not exit in $[a \dots a_1 - 1]$, S' exits on the \$ marker. This tells M' to do a dummy pull to save the state q that S was in when S' hit the \$, and then to execute a put that copies y_1 from the pull bays to the put bays rightward from cell a. M' then effects $a := 2a$ so now $a = a_1$, lets $a_2 := 2a_1$, pokes another \$ to the left of cell a_2, pulls $y_2 := S'_q[a_1 \dots a_2 - 1]$ to the pull bays, and then puts y_2 into the put bays rightward of cell a_1. Since the \$ endmarker is in cell $Na_1 - 1$, this move is valid; nor does y_2 overlap y_1. If S didn't halt in $[a_1 \dots a_2 - 1]$, M' saves the state q' that S was in when S' hit cell a_2, setting things up for the next stage with $a_3 := 2a_2$. The process is iterated until S finally exits in some cell b in some interval $[a_{j-1} \dots a_j - 1]$. Then $y := y_1 y_2 \cdots y_j$ equals $S[a \dots b]$. M' moves to cell Na_j, puts y onto the extension track rightward of cell Na_j, pulls y to the buffer track, and "cleans up" the extension track as before. M' then takes $(K+1)$-many steps backward to cell a_{j-1} and cleans up the pull and put bays with a pull and a put. Finally M' effects $a := \lfloor a/2 \rfloor$ until it finds the $\wedge$ originally placed at cell a, meanwhile removing all of the \$ markers, and then removes the $\wedge$. This completes the simulated pull by S.

Let j be such that $a_j \leq b < a_{j+1}$. Then the number m of symbols read by S is at least $a_j - a$. An induction on j shows that the running totals of both w' and μ-acc' stay bounded by Dm, where D is a constant that depends only on M, not on a or j. Hence the μ-time for the simulation by M' is within $2D$ times the μ-time charged to M for the pass. (However, when $j > 0$, μ-acc'/μ-acc may no longer be bounded by a constant.)

(b) $0B$. M' first runs S on cell 0 only and stores the output y_0 on the first cells of the C-many put bays. M' then follows the procedure for *RaB* with $a = 1$. The analysis is essentially the same.

(c) *LaB*. M' first pokes a $\wedge$ to the left of cell a and \$ to the left of cell $\lfloor a/2 \rfloor$. The $\wedge$ allows M' to detect whether a is even or odd; i.e., whether it needs to simulate $a := 2a$ or $a := 2a+1$ to recover cell a. M' then pulls $y_1 := S'[a \dots \lfloor a/2 \rfloor]$ to the pull bays. Note that cell $\lfloor a/2 \rfloor$ is included; M' avoids a crash by remembering the first $2C$-many symbols of y_1 in its finite control. If S didn't exit in $[a \dots \lfloor a/2 \rfloor]$, M' remembers the state q that S would have gone to after processing cell $\lfloor a/2 \rfloor$. M' then copies cells $[0 \dots \lfloor a/2 \rfloor - 1]$ of the main track into cells $[\lfloor a/2 \rfloor + 1 \dots a]$ of the holding track, and does a leftward pull by S'_q to finish the work by S, stashing its output y_2 on the put bays. If S'_q does not exit before hitting the \$, then S ran off the left end of the tape and M crashed. Let $y := y_1 y_2$. Since $|y| \leq Ca$, M' can copy y to the buffer via cell Na of the extension track by means similar to before, and "clean up" the pull and put bays and holding and extension tracks before returning control to cell a. Here both w' and μ-acc' stay within a fixed constant factor of the corresponding charges to M for the pass.

(d) *BaR*. M' marks cell a on the left with a \$, and does a dummy simulation of S on cells $[0 \dots a-1]$ of the buffer track. If S exits in that interval, M' puts $S[0 \dots a-1]$ directly onto the main track, and this completes the simulated pass. If not, M' puts $y_0 := S[0 \dots a-1]$ onto the holding track rightward of cell a, and remembers the state q in which S' hits the \$. M' then follows the procedure for simulating RaB beginning with S'_q, except that it copies $@^a y_0 y_1 \cdots y_j$ to the extension track via cell Na_j. The final pull then goes to the main track but translates @ by B so that the output written by M lines up with cell a of the main track. There is no need to "clean up" the read portion of the buffer tape since all writes to it are delimited. A calculation similar to that for RaB yields a constant bound on the μ-time and work for the simulated pass, though possibly not on the μ-access charges.

(e) $B0$. Under the simulation, this is the same as $0B$ with the roles of the main track and buffer track reversed, and @ translated to B.

(f) BaL. M' marks cell a on the left with \$ and puts $y_1 := S[0 \dots a{-}1]$ *rightward* from cell a of the holding track. If S exits in that interval of the buffer tape, M' then pulls y_1 to the left end of the holding track. Note that if $|y_1| > a{+}1$ then M was about to crash. M' remembers the first symbol c' of y_1 in its finite control to keep this last pull valid just in case $|y_1| = a{+}1$. Then M' puts c' into cell a, pokes a \$ to the left of cell $\lfloor a/2 \rfloor$, and executes a "delay-1 copy" of the holding track up to the \$ into the main track leftward from cell a. If a B or @ is found on the holding track before the \$, meaning that $|y_1| \leq \lfloor a/2 \rfloor$, the copy stops there and the simulated BaL move is finished. If not, i.e., if $|y_1| > \lfloor a/2 \rfloor$, then the delay allows the character c'' in cell $\lfloor a/2 \rfloor - 1$ of the holding track to be suppressed when the \$ is hit, so that the copy is valid. Since $|y_1| > \lfloor a/2 \rfloor$, M' can now afford to do the following: poke a \$ to the *right* of cell a, effect $a := 2a$, and do a *leftward pull* of cells $[2a \dots a{+}1]$ of the holding track into cells $[0 \dots a-1]$ of the main track, translating @ as well as B by B to leave previous contents of the main track undisturbed. This stitches the rest of y_1 beginning with c'' correctly into place. M' also cleans up cells $[0 \dots 2a]$ of the holding track by methods seen before, and removes the \$ signs.

If S does not exit in $[0 \dots a{-}1]$, M' executes a single Ra move starting S' from cell a, once again holding back the first character of this output y_2 just in case y_1 was empty and $|y_2| = a{+}1$. If this pull is invalid then likewise $|y_2| > a{+}1$ and M crashed anyway. M' then concatenates y_2 to the string y_1 kept on the holding track to form y, and does the above with y. As in LaB, the overhead in both w and μ-*acc* is constant. This completes the proof. ☐

The converse simulation of a BM by a BM with buffer is clear and has constant-factor overheads in all measures, by remarks following Definition 3.1. It is interesting to ask whether the above can be extended to a linear simulation of a *concatenable* buffer (cf. [46]), but this appears to run into problems related to the nonlinear lower bounds for the Touch Problem in [2]. The proof gives $w'(n) = O(w(n) + n)$ and $R'(n) = O(R(n) \log s(n))$. For μ-*acc*$'$, the charges in the rightward moves are bounded by a constant times $\sum_{j=0}^{\log b} \mu(b/2^j)$. For $\mu = \mu_d$ this sum is bounded by $2d\mu_d(b)$, and this gives a constant-factor overhead on μ_d-*acc*. However, for $\mu = \mu_{\log}$ there is an extra factor of $\log b$.

COROLLARY 4.2. *A BM that violates the validity conditions on passes can be simulated linearly by a BM that observes the restrictions.* ☐

We digress briefly to show that allowing simultaneous read and overwrite on the main tape does not alter the power of the model, and that the convention on B gives no power other than *shuffle*. A *two-input Mealy Machine* (2MM) is essentially the same as a 2-input GST with $\rho : (Q \setminus F) \times \Gamma^2 \to \Gamma^*$.

Proposition 4.3. *Let M be a BM with the following extension to the buffer mechanism: in a put step, M may invoke any 2MM S that takes one input from the buffer and the other from the main tape, writes to the main tape, and halts when the buffer is exhausted. Then M can be simulated by a BM M' with buffer at a constant-factor overhead in all measures, for all μ.*

Proof. To simulate the put by a 2MM S, M' copies the buffer to a separate track so as to interleave characters with the segment of the main tape of M concerned. Then M' invokes a GST S' that takes input symbols in twos and simulates S. Finally M' copies the output of S' from its own buffer over the main tape segment of M. □

Proposition 4.4. *At a constant-factor overhead in all measures, for all μ, a BM M can be simulated by a BM M' that lacks B but has the following implementation of shuffle: M' has the above buffer extension, but restricted to the fixed 2MM which interleaves the symbols of its two input strings.*

Proof. Let Γ' consist of Γ together with all ordered pairs of characters from Γ; then the fixed 2MM can be regarded as mapping $\Gamma^* \times \Gamma^*$ onto Γ'^*. Now consider any GST S of M that can output blanks. Let S' write a dummy character @ in place of B, and let M' shuffle the output of S' with the content of the target block of the main tape. Finally M' executes a pass which, for all $c_1, c_2 \in \Gamma$ with $c_1 \neq @$, translates $(c_1, @)$ to c_1 and (c_1, c_2) to c_2. □

Besides the tracking property, our further simulations require something which, again for want of a standard mathematical name, we call the following:

Definition 4.1. A memory access cost function μ has the *tape compression property* if $(\forall\epsilon > 0)(\exists\delta > 0)(\forall^\infty a)\,\mu(\lceil\delta a\rceil) < \epsilon\,\mu(a)$.

Lemma 4.5. *For any $d \geq 1$, the memory cost function μ_d has the tape compression property. In consequence, $\sum_{i=0}^{\log_2 b} \mu_d(\lceil b/2^i\rceil) = O(\mu_d(b))$.*

Proof. Take $\delta < \epsilon^d$. If δ of the form $1/2^k$ satisfies (a) for $\epsilon := 1/2$, then by elementary calculation, for all but finitely many b, $\sum_{i=0}^{\log_2 b} \mu(\lceil b/2^i\rceil) \leq 2k\mu(b)$. □

Lemma 4.5 promises a constant-factor overhead on the memory-access charges for "staircases" under μ_d, whereas an extra log factor can arise under $\mu_{\log}$. The simulation of random access by tree access in the next lemma is the lone obstacle to extending the results that follow to $\mu_{\log}$. Since any function $\mu(m)$ with the tape compression property must be $\Omega[m^\epsilon]$ for some $\epsilon > 0$, this pretty much narrows the field to the functions μ_d. To picture the tree we write Up, Down Left, and Down Right in place of the moves $\lfloor a/2\rfloor$, $2a$, and $2a+1$ by M'.

Lemma 4.6. *For every BM M with address mechanism, there is a basic BM M' such that for all $d \geq 1$, M' simulates M linearly under μ_d.*

Proof. We need to show how M' simulates a load step of M that loads an address a_1 from cells $[a_0 \ldots b_0]$ of the main tape. Let $m := |a_0 - b_0| + 1$. M' makes one spare track for operations on addresses. M' first pulls a_1 in binary to the left end of this track. By Theorem 4.1 we may suppose that this pull is valid. The cost is proportional to the charge of $\mu(a_0) + m + \mu(b_0)$ to M for the load. By our convention on addresses, the least significant bit of a_1 is leftmost. In this pull, M' replaces the most significant '1' bit of a_1 by a '\$' endmarker. M' then moves Up until its cell-a head reaches cell 1. With $k := \lceil\log_2 a_0\rceil$, the total memory access charges so far are proportional to $\sum_{i=0}^{k} \mu(2^i)$, which is bounded by a fixed constant times $\mu(a_0)$ by

Lemma 4.5. Since the number of bits in a_1 is bounded by Cm, where C depends only on M, the work done by M' is bounded by $2Cm + k$. Since $k < \mu(a_0)$, we can ignore k. Hence the μ-time charged so far to M' is bounded by a fixed constant of that charged to M for the load.

M' now executes a rightward pull that copies all but the bit b before the \$ endmarker, b being the second most significant bit of a_1. This pull is not valid owing to an overlap on the fresh track, but by Corollary 4.2 we may suppose that it is valid. If $b = 0$ M' moves Down Left, while if $b = 1$, M' moves Down Right. M' then executes a put that copies the remainder of a_1 (plus the \$) rightward from the new location a. M' iterates this process until all bits of a_1 are exhausted. At the end, $a = a_1$. Because of the tracking, M' moves Down Left once more so that it scans cell $2a$, which is cell a of the main track. This completes the simulated load. Recalling $|a_1| \leq Cm$, and taking $l := \lceil \log_2(a_1) \rceil$, the μ-time for this second part is bounded by a constant times

$$\sum_{i=0}^{l} \mu(2^i) + 2(m-i) + \mu(2^i + 2(m-i)). \tag{3}$$

By Lemma 4.5, the total memory access charges in this sum are bounded by a fixed constant times $\mu(a_1)$. The work to simulate the load is proportional to $m \cdot l$, that is, to $(\log a_1)^2$, which causes an extra log factor over the work by M in the load. The key point, however, is that since M loaded the address a_1, M will be charged $\mu_d(a_1)$ on the next pass, which is asymptotically greater than $(\log a_1)^2$. Hence the μ_d-time of M' stays proportional to the μ_d-time of M. □

Corollary 4.7. *For every BM M with both the address and buffer mechanism, we can find a basic BM M', and a BM M'' with the limited buffer mechanism, such that for any $d \geq 1$, M' and M'' simulate M linearly under μ_d.*

Proof. The constructions of Lemma 4.6 and Theorem 4.1 yield M'. For M'', we may first suppose that M is modified so that whenever M loads an address a, it first stores a spare copy of a at the left end of a special track. Now consider a pass of type $B0$ or $0B$ made by M. M'' invokes a GST that remembers cell 0 and writes 1 to the address tape. Then with $a' = 1$, M'' simulates the pass by a $Ba'R$ or $Ra'B$ move. M'' then recovers the original address a by loading it from the track. Thus far M'' is a BM with address and buffer that doesn't use its cell 0 head. The method of Lemma 4.6 then removes the address mechanism in a way unaffected by the presence of the buffer. □

We remark that Lemma 4.6 and Theorem 4.1 apply to different kinds of pass by M, with two exceptions: First, pulling 1 to the left end of the track in the proof of Lemma 4.6 may require simulating a buffer. However, this can be accounted against the cost to M for the load. Second, the buffer is needed for overlaps in the further processing of a_1. However, this is needed for at most $O(\log\log(a_1))$-many passes, each of which involves $O(\log a_1)$ work, and these costs are dominated by the time to process a_1 itself. Hence in Corollary 4.7 the bounds from Lemma 4.6 and Theorem 4.1 are additive rather than compounded, and with $\mu = \mu_d$ we obtain for M', $\mu_d\text{-}acc'(n) = O(\mu_d\text{-}acc(n))$, $\mu_{\log}\text{-}acc'(n) = O(\mu_{\log}\text{-}acc(n) \log s(n))$, $w'(n) = O(w(n) + n + R(n) \log s(n))$, and $R'(n) = O(R(n) \log s(n))$.

Lemma 4.8. *For every RAM-BM M, we can find a BM M' with the address and buffer mechanisms, such that for any memory cost function μ that is $\Omega(\log n)$, M' simulates M linearly under μ.*

Proof. First, M' makes separate tracks for the address tapes and worktapes of M, and also for storing the locations of the heads on these tapes of M. Whenever M begins a block move $S\,[a_1 \ldots b_1]$ *into* $[a_2 \ldots b_2]$, M' first computes the signs of $b_1 - a_1$ and $b_2 - a_2$, and remembers them in its finite control. M' then loads the true address of cell a_1 on the main tape, and pulls the data through a copy of S labeled RaB or LaB—depending on sign—to the buffer. Then M' loads 0 to access a_2, loads a_2 itself, and finally copies the buffer right or left from cell a_2. Since μ is $\Omega(\log n)$, the μ-time charged to M' is bounded by a fixed constant times the charge of $1 + |b_1 - a_1| + \max\{\,\mu(a_1), \mu(a_2), \mu(b_1), \mu(b_2)\,\}$ incurred by M. Similarly the μ-*acc* charge to M' has the same order as that to M, though if $|b_1 - a_1| < \log(b_1)$, this may not be true of the work.

If M executes a standard RAM-TM transition, the cost to M is $1 + \mu(a_1) + \mu(c)$, where a_1 is the cell addressed on the main tape and c is the greatest extent of an address tape or worktape head of M. M' first loads a_1 and writes the symbol written by M into location a_1 with a unit put. Then M' loads each of the addresses for the other tapes of M in turn, updates each one with a unit pull and a unit put, remembers the head movement on that tape, and increments or decrements the corresponding address accordingly. The time charge for updating the other tapes stays within a fixed constant factor of $\mu(c)$. ☐

Remark: It would be nice to have the last simulation work when the charge to M for a RAM-TM transition is just $1 + \mu(a_1)$. The difficulty is that even though $|a_1| < a_1$, it need not hold that $c < a_1$, since M might be using a lot of space on its worktapes. The issue appears to come down to whether a multitape TM running in time t can be simulated by a BM in μ-time $O(t)$. We discuss related open problems in Section 8.

LEMMA 4.9. *A finger BM can be simulated by a BM with address and buffer mechanisms, with the same bounds as in Lemma 4.8.*

Proof. M' stores and updates the finitely-many markers on separate tracks in a similar manner to the last proof. The extra work per block move simulated to write or load these addresses is $O(\log s(n))$ as before. Both here and in Lemma 4.8, $R'(n) = O(R(n))$. ☐

THEOREM 4.10. *Let M be a RAM-BM, a finger BM, or a BM with the address and/or buffer mechanisms. Then we can find a BM M' that simulates M linearly under any μ_d.*

Proof. This follows by concatenating the constructions of the last two lemmas with that of Corollary 4.7. Since $R'(n) = O(R(n))$ in the former, the bounds on work and pass count remain $w'(n) = O(w(n) + n + R(n)\log s(n))$ and $R'(n) = O(R(n)\log s(n))$. ☐

This completes the simulation of most of the richer forms of the model by the basic BM, with a constant factor overhead in μ_d-*time*. By similar means, one can reduce the number of markers in a finger BM all the way to four. In going up to the richer forms, we encounter the problem that the finger BM and RAM-BM have pre-set block boundaries for input, and if the strict boundary condition is enforced, also for output.

4.2. Simulations for pre-set block boundaries. The simulation in Theorem 4.1 does not make M' self-delimiting because it does not predetermine the cell $b \in [a_0 \dots a_1]$ in which its own simulating GST S' will exit. We could try forcing S' to read all of $[a_0 \dots a_1]$, but part (a) of the proof of Theorem 4.1 had $a_1 := 2a_0$, and if e.g. $\mu(a_0) = \sqrt{a_0}$ and $b - a$ is small, M' would do much more work than it should. However, if one chooses the initial increment e to be too small in trying $a_1 := a_0 + e$, $a_2 := a_0 + 2e$, $a_3 := a_0 + 4e \dots$, the sum of the μ-access charges may outstrip the work. To balance the charges we take $e := \mu(a_0)$. This requires M' to *calculate* $\mu(a)$ dynamically during its computation, and involves a concept of "time-constructible function" similar to that defined for Turing machines in [36].

DEFINITION 4.2. Let μ be a memory cost function, and let $t : \mathbf{N} \to \mathbf{N}$ be any function. Then t is *μ-time constructible* if $t(n)$ is computable in binary notation by a BM M in μ-time $O(t(n))$.

Note that the time is in terms of n, not the length of n. We use this definition for $t = \mu$ itself, in saying that μ is μ-time constructible. The following takes d to be rational because there are real numbers $d \geq 1$ such that no computable function whatever gives $\lceil m^{1/d} \rceil$ to within a factor of 2 for all m. In this section it would suffice to estimate $\lceil m^{1/d} \rceil$ by some binary number of bit-length $|m|/d$, but we need the proof idea of incrementing fingers and the exact calculation of $\mu_d(m)$ for later reference.

LEMMA 4.11. *For any rational $d \geq 1$, the memory cost function μ_d is μ_d-time constructible by a finger BM that observes the strict boundary condition.*

Proof. For any rational $d \geq 1$, the function $\lceil m^{1/d} \rceil$ is computable in polynomial time, hence in time $(\log m)^{O(1)}$ by a single-tape TM T. The finger BM M simulates the tape of T beginning in cell 2, and tracks the head of T with its "main marker" m_1. M also uses a character @ which is combined into others like so: if T scans some character c in cell a, M scans $(c, @)$. M then uses two unit block moves $S[a \dots a]$ *into* $[0 \dots 0]$ and $S[0 \dots 0]$ *into* $[a \dots a]$ to read and write what T does. It remains to simulate the head moves by T.

To picture a tree we again say UP, DOWN LEFT, and DOWN RIGHT in place of moves from a to $\lfloor a/2 \rfloor$, $2a$, or $2a{+}1$. M can test whether a is a left or right child by moving UP and DOWN LEFT and seeing whether the character scanned contains the @. If T moves right and a is a left child, M then intersperses moves UP and DOWN RIGHT with unit block moves to and from cell 0 to change $(c, @)$ back to c and place @ into cell $a{+}1$. If instead a is a right child, M introduces a new marker m_5 into cell 1, and writes $\wedge$ there. M moves m_5 DOWN LEFT to count how far UP m_1 has to go until it reaches either a left child or the root (i.e., cell 1). By unit block moves, M carries @ along with m_1, and by assigning a finger to marker m_5, can test whether m_5 is on cell 1. If m_1 reaches a left child, M moves it UP, DOWN RIGHT, and then DOWN LEFT until m_5 comes back to the $\wedge$. Then m_1 is in cell $a{+}1$. If m_1 hits the root marked by $\wedge$, then a had the form $2^k - 1$, and so M moves m_1 DOWN LEFT k times. The procedure for decrementing m_1 when T moves left is similar, with RIGHT and LEFT reversed.

For each step by T, the work by M is proportional to $\log a$. By Lemma 4.5 for μ_d, the total memory-access charge for incrementing or decrementing a finger in cell a is $O(\mu_d(a))$. Since $a \leq (\log m)^{O(1)}$, the total μ_d-time for the simulation is still a polynomial in $\log m$, and hence is $o(\mu_d(m))$. $\square$

This procedure can also be carried out on one of 2^K-many tracks in a larger machine, computing $a \pm 2^K$ instead of $a \pm 1$ to follow head moves by T. The counting idea of the next lemma resembles the linear-size circuits constructed for 0-1 sorting in [55].

LEMMA 4.12. *The function* $\#\mathrm{a}(x)$*, which gives the number of occurrences of 'a' in a string* $x \in \{\mathrm{a}, \mathrm{b}\}^*$*, is computable in linear* μ_1*-time by a BM that observes the strict boundary condition.*

Proof. The BM M operates two GSTs S_1 and S_2 that read bits of x in pairs. Each records the parity p of the number of pairs 'ab' or 'ba' it has seen thus far, and if $|x|$ is odd, each behaves as though the input were xb. S_1 outputs the final value of p to a second track. S_2 makes the following translations

$$\mathrm{aa} \mapsto \mathrm{a} \qquad \mathrm{bb} \mapsto \mathrm{b} \qquad \mathrm{ab,ba} \mapsto \begin{cases} \mathrm{b} & \text{if } p = 0 \\ \mathrm{a} & \text{if } p = 1 \end{cases}$$

to form a string x' such that $|x'| = \lceil |x|/2 \rceil$. Then $\#\mathrm{a}(x) = 2\#\mathrm{a}(x') + p$. This is iterated until no a's are left in x, at which point the bits p combine to form $\#\mathrm{a}(x)$ in binary notation with the least significant bit first.

M begins with one marker m_1 in cell $n{-}1$. We first note that even setting up the two tracks requires a trick to get two more markers to cell $n-1$. M starts a marker m_5 in cell 1 and moves it DOWN LEFT or DOWN RIGHT according to whether m_1 is on a left or right child. When m_1 reaches cell 1, m_5 records $n-1$ in reverse binary notation. Then M starts moving m_5 back up while ferrying m_2, m_3 along with m_1. Then M places m_2 and m_3 into cells $2n$ and $4n{-}1$, and with reference to Example 2.1, executes $(c \mapsto c@)[0 \ldots n{-}1]$ *into* $[2n \ldots 4n{-}1]$ and *copy*$[2n \ldots 4n{-}1]$ *into* $[0 \ldots 2n{-}1]$. This also uses one increment and decrement of a marker as in the proof of Lemma 4.11.

M uses a new marker m_6 to locate where the next bit p will go, incrementing m_6 after running S_1. In running S_2, always $|S_2(x')| = \lceil |x'|/2 \rceil$, and by appropriate parity tests using its markers m_1, m_2, and m_3, M can place its fingers so that all these moves are valid and meet the strict boundary condition. For the kth iteration by S_2, these three markers are all on cells with addresses lower than $n/2^{k-2}$, and even if each needs to be incremented by 1 with the help of m_5, the μ_1 charges for simulating the iteration still total less than a fixed constant times $n/2^{k-2}$. This also subsumes the $O(\log^2 n)$ charge for updating m_6. Hence the sum over all iterations is still $O(n)$. □

THEOREM 4.13. *For every BM* M *and rational* $d \geq 1$*, we can find a finger BM* M' *that simulates* M *linearly under* μ_d *and observes the strict boundary condition.*

Proof. As in the proof of Theorem 4.1, let C be the maximum number of characters output in any GST transition of M, and let $K := \log_2(2C + 6)$. M' first makes $N := 2^K$ tracks, by using the last proof's modification of the procedure of Example 2.1. Besides $2C$-many tracks for handling the output of passes and one track for the main tape of M, M' uses one track to record the current address a of M with the least significant bit *rightmost*, one to compute and store $e := \mu_d(a)$ via Lemma 4.11, one to store addresses a_j below, two for Lemma 4.12, and one for other arithmetic on addresses. M' uses eight markers. Marker m_1 occupies cell Na to record the current address a of M. A move to $\lfloor a/2 \rfloor$ by M is handled by moving m_1 UP $K{+}1$ times and DOWN LEFT K times, and other moves are handled similarly. Meanwhile, marker m_6 stays on the last bit of the stored address a, and updating a requires only one marker increment or decrement and $O(\log \log a)$ work overall. From here on we suppress the distinction between a and Na and other details that are the same as in Theorem 4.1.

First consider a rightward pull by M that starts a GST S from cell a_0 on its main tape. M' has already stored a_0 in binary, and computes $e := \mu_d(a_0)$. Since $\mu_d(a_0) \leq a_0$, e fits to the left of marker m_6 in cell $|a|$. M' then places m_3 into cell $|a|+1$ and m_4 into cell $2|a|+1$, and executes two block moves from $[|a|\dots 0]$ and $[0\dots|a|]$ into $[|a|+1\dots 2|a|+1]$ that shuffle a_0 and e on their respective tracks with the least significant bits aligned and leftmost. M' then executes *add* $[|a|+1\dots 2|a|+1]$ *into* $[|a|\dots 0]$ to produce a_1. A final carry that would make $|a_1| > |a_0|$ and cause a crash can be caught and remembered in the finite control of M' by running a "dummy addition" first, and then marking cell $2|a|+1$ to suppress its output by the GST *add*. Then M' "walks" marker m_2 out to cell a_1 by using m_3 to read the value of a_1 and m_5 to increment m_3.

Next M' walks m_4 out to cell e (i.e., Ne), and keeps m_3 in cell 0. Let S' be a copy of S which pads the output of each transition out to length exactly C, and which sends its output z to the C-many tracks used as "pull bays." S' is also coded so that if S exits, S' records that fact and writes $@^C$ in each transition thereafter. Then M' can execute $S'[a_1\dots a_2]$ *into* $[0\dots e]$ in compliance with the strict boundary condition. Now M' can calculate the number i of non-@ symbols in z by the method of Lemma 4.12. To write the true output $y_1 = S[a_0\dots a_1]$ and ensure the block move is valid, M' must still use the pull bays to hold y_1, so M' calculates $i' := \lceil i/C \rceil$ (actually, $i' = N\lceil i/C \rceil$). Next M walks m_4 out to cell i', and can finally simulate the first segment of the pass by S by executing $S[a_1\dots a_2]$ *into* $[0\dots i']$.

If S exited in $[a_0\dots a_1]$, M' need only transfer the output y_1 of the last pass onto the left end of the main track. This can be done in two block moves after locating markers into cells i', Ci', and $2Ci'$. Else, M' transfers y_1 instead to the put bays and assigns a new marker m_7 to the "stitch point" in the put bays for the next segment of y. The final marker m_8 goes to cell a_1 and is used for the left-end of the read block in all succeeding segments. In three block moves, M' can both double e to $2e$ and compute $a_2 := a_0 + 2e$ using *add* as before. If and when the current value of e has length greater than $|a_0|$, M' reassigns marker m_6 to the end of e rather than a_0, incrementing it each time e is doubled. Then M' walks m_2 out to cell a_2 and, remembering the state q of S where the previous segment left off, produces $y_2 := S_q[a_1{+}1\dots a_2]$ by the same counting method as before. To stitch y_2 into place on the put bays, M' converts the current location of m_7 into a numeric value k, adds it to $i := |y_2|$, and finds cells $i+k$ and $2i+k$ for two block copies. In case S did not exit in $[a_1\dots a_2]$, m_7 is moved to cell $i+k$, m_8 to a_2, m_2 to $a_3 := a_0 + 4e$, and the process is repeated.

Let b be the actual cell in which S exits, and let $j \geq 0$ be such that $a_j < b \leq a_{j+1}$. Then the μ_d-time charged to M for the pull is at least

$$t_j := \mu_d(a_0) + \mu_d(a_0 + 2^{j-1}e) + 2^{j-1}e \geq 2e + 2^{j-1}e. \tag{4}$$

(For $j = 0$, read "2^{j-1}" as zero.) By Lemma 4.5, the memory access charge for walking a marker out to cell a_j is bounded by a constant (depending only on d) times $\mu_d(a_j)$. The charges for the marker arithmetic come to a polynomial in $\log a_j$, and the charges for stitching segments y_j into place stay bounded by the work performed by M'. Hence the μ_d-time charged to M' is bounded by a constant times

$$u_j := \mu_d(a_0) + \sum_{i=0}^{j} \mu_d(a_0 + 2^i e) + e + \sum_{i=0}^{j-1} 2^i e. \tag{5}$$

Then $u_j \leq e + \sum_{i=0}^{j} \mu_d(2^{i+1}a_0) + 2^j e \leq 2^{j+2}e + 2^j e \leq 10t_j$.

For a leftward pull step by M, M' uses the same choice of $e := \mu_d(a_0)$. If $e > a_0/2$, then M' just splits $[0 \ldots a_0]$ into halves as in the (LaB) part of the proof of Theorem 4.1. Else, M' proceeds as before with $a_{j+1} := a_0 - 2^j e$, and checks at each stage whether $a_{j+1} \geq a_0/2$ so that the next simulated pull will be valid. If not, then the amount of work done by M thus far, namely $2^{j-1}e$, is at least $a_0/4$. Thus M' can copy all of $[a_j \ldots 0]$ to another part of the tape and finish it off while remaining within a constant factor of the charge to M. The remaining bounds are much the same as those for a rightward pull above.

For a rightward or leftward put, marker m_1 is kept at the current address a, cell 0 is remembered in the finite control, and the procedure for a rightward pull is begun with $a_0 = 1$ and m_8 assigned there. Here $e = 1$, and the rest is a combination of the (BaR) or (BaL) parts of the proof of Theorem 4.1 to ensure validity, and the above ways to meet the strict boundary condition in all block moves. □

Remarks: This simulation can be made uniform by providing d as a separate input. It can also be done using 8 tracks rather than $2C + 6$, though even taking $e := \mu_d(a_0)/C$ does not guarantee that the *third* stage of a rightward pull, which reads $[a_0 + 2e, a_0 + 4e]$, will be valid. The fix is first to write the strings y_j further rightward on the tape, then assemble them at the left end. Theorem 4.13 preserves $w(n)+\mu_d\text{-}acc(n)$ up to constant factors, but doesn't do so for either $w(n)$ or $\mu_d\text{-}acc(n)$ separately. When $d < 1$, the case $b = a$ gives a worst-case extra work of $a^{1/d}$, while the case of $b = 2a$ gives a total memory access charge of roughly $2(\log a)(d-1)/d$ times $\mu_d(a)$. This translates into $w'(n) = O(w(n)+n+R(n)s(n)^{1/d})$ and $\mu_d\text{-}acc'(n) = O(\mu_d\text{-}acc(n) \log s(n))$. However, when $d = 1$, *both* w and $\mu_1\text{-}acc$ are preserved up to a factor of $10N$. Allowing that $\mu_d(a_0)$ can be estimated to within a constant factor in $O(\log a_0)$ block moves, the pass count still carries $R'(n) = O(R(n) \log^2 s(n))$ because each movement in walking a marker to a_j adds 1 to R'. The following shows some technical improvements of having addressing instead of tree access.

THEOREM 4.14. *Let $\mu = \mu_{\log}$ or $\mu = \mu_d$ with d rational. Then every BM M can be simulated linearly under μ by a RAM-BM M' with address loading that observes the strict boundary condition.*

Proof. For μ_d the simulation of the finger BM M' from the last proof by a RAM-BM is clear—the RAM-BM can even use RAM-TM steps for the address arithmetic. For $\mu_{\log}$, the point is that M' can take $e := |a_0|$, and we may presume e is already stored. The calculated quantities a_j can be loaded in one block move. (Using RAM-TM steps to write them would incur $\mu_{\log}$ access charges proportional to $\log a_0 \log\log a_0$.) The tradeoff argument of the proof of Theorem 4.13 works even for $\mu_{\log}$, and the above takes care of a constant-factor bound on the other steps in the simulation. This also gives $R'(n) = O(R(n) \log s(n))$. □

The tradeoff method of Theorem 4.13 seems also to be needed for the following "tape-reduction theorem."

THEOREM 4.15. *For every rational $d \geq 1$, a multitape BM M can be simulated linearly in μ_d-time by a one-tape BM M'.*

Proof. Suppose that M uses k tapes, each with its own buffer, and GSTs S that produce k output strings as well as read k inputs. We first modify M to a machine M' that has k main tracks, k address tracks, one "input track," and one "buffer track." For any pass by M with S, M' will interleave the k inputs on the input track, do one separate pull for each of the k outputs of S, and interleave the outputs on its buffer track. When M subsequently invokes a k-input GST T to empty its buffers, M' uses a 1-tape GST that simulates T on the buffer track, invoking it k times to write each of the k outputs of T to their destinations on the main tracks.

It remains only to show how M' marks the portions of the inputs to interleave. As in the proof of Theorem 4.13, there is the difficulty of not knowing in advance how long S will run on its k inputs. The solution is the same. M' first calculates the maximum a_j of the addresses $a_1, \ldots, a_k$ on its address tracks, and then calculates $e := \mu_d(a_j)$. For each i, $1 \leq i \leq k$, M' drops an endmarker into cell $a_i \pm e$ according to the direction on main track i. Then M' copies only the marked-off portions of the tracks, putting those on its input track, and simulates the one-tape version S_1 of S. If S_1 exits within that portion, then M' continues as M' does. If S_1 does not exit within that portion, M' tries again with $a_i \pm 2e$, $a_i \pm 4e, \ldots$ until it does. The same calculation as in Theorem 4.13, plus the observation that if the direction on track j is leftward then no track uses an address greater than $2a_j$, completes the proof. $\square$

Finally we may restate the Main Robustness Theorem 3.1 in a somewhat stronger form:

THEOREM 4.16. *For any rational $d \geq 1$, all of the models defined in Section 3 are equivalent, linearly in μ_d-time, to a BM in reduced form that is self-delimiting with '\$' as its only endmarker.*

Proof. This is accomplished by Theorems 2.1 through 4.15. The procedures of Lemmas 4.13 and 4.6 and Theorem 4.1 are self-delimiting, and need only one endmarker \$. The trick of writing \$ on special tracks into the cell immediately left or right of the addressed cell a allows \$ to survive the proof of Theorem 2.1 without being "tupled" into the characters c_0, c_1, or c_a. $\square$

With all this said and verified, we feel justified in claiming that there is one salient Block Machine model, and that the formulations given here are natural. The basic BM is the tightest for investigating the structure of computations, and helps the lower bound technique we suggest in Section 8. The richer forms make it easier to show that certain functions do belong to $\mathrm{D}\mu_d\mathrm{TIME}[t(n)]$.

5. Linear Speed-Up and Efficiency. The following "linear speed-up" theorem shrinks the constants in all the above simulations, at the usual penalty in alphabet size. First we give a precise definition:

DEFINITION 5.1. The *linear speed-up property* for a model of computation and measure of time complexity states that for every machine M with running time $t(n)$, and every $\epsilon > 0$, there is a machine M' that simulates M and runs in time $\epsilon \cdot t(n) + O(n)$.

In the corresponding definition for Turing machines in [36], the additive $O(n)$ term is $n+1$ and is used to read the input. For the DTM, time $O(n)$ properly contains time $n+1$, while for the NTM these are equal [13]. For the BM under cost function μ, the $O(n)$ term is $n + \mu(n)$.

THEOREM 5.1. *With respect to any unbounded memory cost function μ that has the tape compression property, all of the BM variants described in Sections 2 and 3 have the linear speed-up property.*

Proof. Let the BM M and $\epsilon > 0$ be given. The BM M' uses two tracks to simulate the main tape of M. Let δ in the tape compression property be such that for almost all n, $\mu(\delta n) \leq (\epsilon/12C) \cdot \mu(n)$. Here C is a constant that depends only on

M. Let $k := \lceil 1/2\delta \rceil$, let '@' stand for the blank in Γ, and let $\Gamma' := \Gamma^k \cup \{ B \}$. M' uses B only to handle its own two tracks. We describe M' as though it has a buffer; the constant C absorbs the overhead for simulating one if M' lacks the buffer mechanism. On any input x of length n, M' first spends $O(n)$ time units on a pull step that writes x into $\lceil n/k \rceil$-many characters over the compressed alphabet Γ' on the main track. Thereafter, M' simulates M with compressed tapes. In any pass by M that writes output to the main tape, M' writes the compressed output to the alternate track. M' then uses the pattern of @ symbols in each compressed output character to mask the elements of each main track character that should not be overwritten, sending the combined output to the buffer. One more pass writes the result back to the main tape. If the cost to M for the pass was $\mu(a) + |b-a| + \mu(b)$, the cost to M', allowing for the tracking, is no more than

$$\begin{aligned} &3\,[\,\mu(2\lceil a/k \rceil) + (2/d)|b-a| + 2 + \mu(2\lceil b/k \rceil)] \\ &\quad \leq (\epsilon/2)\mu(a) + (\epsilon/2)|b-a| + 6 + (\epsilon/2)\mu(b). \end{aligned}$$

The '+2' and '+6' allow for an extra cell at either end of the compressed block. Since μ is unbounded, we have $\mu(a)\cdot(\epsilon/2) + 6 \leq \epsilon \cdot \mu(a)$ for all but finitely many a. The main technical difficulty of the standard proof for TMs is averted because μ absorbs any time that M might spend moving back and forth across block boundaries. The compression by a factor of ϵ holds everywhere except for cells $1, \ldots, m$ on the main tape, where m is least such that $\mu(m) \geq 12/\epsilon$, but M' can keep the content of these cells in its finite control. The remaining details are left to the reader. For BMs with address tapes, we may suppose that the addresses are written in a machine-dependent radix rather than in binary. □

Corollary 5.2. *For all of the simulations in Theorems 2.1–4.15, and all $\epsilon > 0$:*
(a) If M runs in μ_d-time $t(n) = \omega(n)$, then M' can be constructed to run in μ_d-time $\epsilon t(n)$ for all but finitely many n.
(b) If M runs in μ_d-time $O(n)$, then M' can be made to run in μ_d-time $(1+\epsilon)n$. □

Mostly because of Lemma 4.6 and Theorem 4.13, the above simulations do not guarantee constant factor overheads in either w or μ-*acc*. They do, however, preserve μ-efficiency.

Proposition 5.3. *For all of the simulations of a machine M by a machine M' in Theorems 2.1–4.15, and memory cost functions μ they hold for, if M is μ-efficient then M' is also μ-efficient.*

Proof. Let K_1 be the constant from the simulation of M by M', and let K_2 come from Definition 2.9(a) for M. Then for all but finitely many inputs x, we have

$$\mu\text{-}time(M', x) \leq K_1(\mu\text{-}time(M, x) + |x|) \leq K_1(K_2(w(M, x) + |x|) \leq 2K_1K_2w(M', x).$$

The last inequality follows because every simulation has $w(M', x) \geq w(M, x)$ and $w(M', x) \geq |x|$. Hence M' is μ-efficient. □

So long as we adopt the convention that every function takes work at least $n+1$ to compute, we can state:

Corollary 5.4. *For any memory cost function μ_d, with $d \geq 1$ and rational, the notion of a language or function being memory efficient under μ_d does not depend on the choice among the above variants of the BM model.* □

We do not have analogous results for parsimony. However, the above allows us to conclude that for $d = 1, 2, 3, \ldots$, memory-efficiency under μ_d is a fundamental property of languages or functions. Likewise we have a robust notion of the class $\mathrm{D}\mu_d\mathrm{TIME}[t(n)]$ of functions computable in μ_d-time $t(n)$, for any time bound $t(n) \geq n$. The next section shows that for any fixed d, the classes $\mathrm{D}\mu_d\mathrm{TIME}[t(n)]$ form a tight hierarchy as the time function t varies.

6. Word Problems and Universal Simulation. We use a simple representation of a list $\vec{x} := (x_1, x_2, \ldots, x_m)$ of nonempty strings in Σ^* by the string $x_1\# \ldots \#x_m\#$, where $\# \notin \Sigma$. More precisely, we make the last symbol c of each element a pair $(c, \#)$ so as to separate elements without adding space, and also use pair characters $(c, @)$ or $(c, \$)$ to mark selected elements. The *size* of the list is m, while the *bit-length* of the list is $n := \sum_{i=1}^{m} |x_i|$. We let r stand for $\max\{\, |x_i| : 1 \leq i \leq m \,\}$. Following [16] we call the list *normal* if the strings x_i all have length r. We number lists beginning with x_1 to emphasize that the x_i are not characters.

LEMMA 6.1.
(a) *The function mark*$(\vec{x}, y)$, *which marks all occurrences of the string y in the normal list $\vec{x}$, belongs to* TLIN.
(b) *The function shuffle*$(\vec{x}, \vec{y})$, *which is defined for normal lists $\vec{x} := (x_1, \ldots, x_m)$ and $\vec{y} := (y_1, \ldots, y_m)$ of the same length and element size r to be $(x_1, y_1, x_2, y_2, \ldots, x_m, y_m)$, belongs to* TLIN. *Here r as well as m may vary.*

Remark: Even if the lists $\vec{x}$ and $\vec{y}$ are not normal, *mark* and *shuffle* can be computed in linear μ_1-time so long as they are *balanced* in the sense that $(\exists k)(\forall i)2^{k-1} < |x_i| \leq 2^k$. This is because a balanced list can be padded out to a normal list in linear μ_1-time (we do not give the details here), and then the padding can be removed. To normalize an unbalanced list may expand the bit-length quadratically, and we do not know how to compute *shuffle* in linear μ_1-time for general lists.

Proof. (a) Let r be the element size of the normal list $\vec{x}$. If $|y| \neq r$, then there is nothing to do. Else, the BM M uses the idea of "recursive doubling" (cf. the section on vector machines in [6]) to produce y^k, where $k = \lceil log_2 m \rceil$. This time is linear as a function of $n = rm$. Then M interleaves $\vec{x}$ and y^k on a separate track, and a single pass that checks for matches between # signs marks all the occurrences of y in $\vec{x}$ (if any).

(b) Suppose m is even. M first uses two passes to divide $\vec{x}$ into the "odd list" $x_1 @^r x_3 @^r \ldots x_{m-1} @^r$ and the "even list" $@^r x_2 @^r x_4 @^r \ldots @^r x_m$. Single passes then convert these to $x_1 @^{3r} x_3 @^{3r} \ldots x_{m-1} @^{3r}$ and $@^{2r} x_2 @^{3r} x_4 @^{3r} \ldots @^{3r} x_m$. A pull step that writes the second over the first but translates @ to B then produces $\vec{x}\,' := x_1 @^r x_2 @^r x_3 @^r \ldots @^r x_m$. If m is odd then the "odd list" is $x_1 @^r x_3 @^r \ldots @^r x_m$ and the "even list" is $@^r x_2 @^r x_4 @^r \ldots @^r x_{m-1} @^r$, but the final result $\vec{x}\,'$ is the same. By a similar process M converts $\vec{y}$ to $\vec{y}\,' := @^r y_1 @^r y_2 \ldots @^r y_m @^r$. Writing $\vec{y}\,'$ on top of $\vec{x}\,'$ and translating @ to B then yields *shuffle*$(\vec{x}, \vec{y})$. This requires only a constant number of passes. $\square$

A *monoid* is a set H together with a binary operation $\circ$ defined on H, such that $\circ$ is associative and H has an element that is both a right and a left identity for $\circ$. We fix attention on the following representation of the *monoid of transformations* $\mathcal{M}_S$ of a finite-state machine S. $\mathcal{M}_S$ *acts* on the state set Q of S and is *generated* by the functions $\{\, g_c : c \in \Sigma \,\}$ defined by $g_c(q) = \delta(q, c)$ for all $q \in Q$, by letting

$\circ$ be composition of maps on Q, and closing out the g_c under $\circ$. Here we ignore the output function ρ of S, intending to use it once the *trajectory* of states S enters on an argument z is computed. We also remark that $\mathcal{M}_S$ need not contain the identity mapping on Q, though it does no harm for us to adjoin it. By using known decomposition theorems for finite transducers [47, 32, 48], we could restrict attention to the cases where each g_c either is the identity on Q or identifies two states (a "reset machine") or each g_c is a permutation of Q and $\mathcal{M}_S$ is a group (a "permutation machine"; cf. [17]). These points do not matter here. We encode each state in Q as a binary string of some fixed length k, and encode each element g of $\mathcal{M}_S$ by the list $q\#g(q)\#\ldots$ over all $q \in Q$. Without loss of generality we extend Q to $Q' := \{0,\ldots,2^k-1\}$ and make g the identity on elements $q \geq n$.

The *word problem* for monoids is: given a list $\vec{g} := g_n g_{n-1} \cdots g_2 g_1$ of elements of the monoid, not necessarily distinct, compute the representation of $g_n \circ g_{n-1} \circ \ldots \circ g_2 \circ g_1$. Let us call the following the *trajectory problem*: given $\vec{g}$ and some $w \in \{0,1\}^k$, compute the n-tuple $(g_1(w), g_2(g_1(w)), \ldots, \vec{g}(w))$. The basic idea of the following is that "parallel prefix sum" is μ_1-efficient on a BM.

LEMMA 6.2. *There is a fixed BM M that, for any size parameter k, solves the word and trajectory problems for monoids acting on $\{0,1\}^k$ in μ_1-time $O(n \cdot k2^k)$. In particular, these problems for any fixed finite monoid belong to* TLIN.

Proof. Let T be a TM which, for any k, composes two mappings $h_1, h_2 : \{0,1\}^k \to \{0,1\}^k$ using the above representation. For ease of visualization, we make T a single-tape TM which on any input of the form $h_2\#h_1\#$ uses only the $2k \cdot 2^k$ cells occupied by the input as workspace, and which outputs $h_2 \circ h_1\#$ shuffled with '@' symbols so that the output has the same length as the input. We also program T so that on input $h\#$, T leaves h unchanged. The running time $t(k)$ of T depends only on k and is $O(k2^k)^2$. As in Example 2.4, we can create a GST S whose input alphabet is the ID alphabet of T, such that for any nonhalting ID I of T, $S(I)$ is the unique ID J such that $I \vdash_T J$.

The BM M operates as follows on input $\vec{g} := g_n\#g_{n-1}\#\cdots\#g_2\#g_1\#$. It first saves $\vec{g}$ in cells $[(nk \cdot 2^k + 1)\ldots(2nk \cdot 2^k)]$ of a separate storage track. We may suppose that n is even; if n is odd, g_n is left untouched by the current phase of the recursion. M first sets up the initial ID of T on successive pairs of maps, viz. $\wedge q_0 g_n\#g_{n-1}\# \wedge q_0 g_{n-2}\#g_{n-3}\#\cdots\wedge q_0 g_2\#g_1\#$. Then M invokes S in repeated left-to-right pulls, until all simulated computations by T have halted. Then M erases all the @s, leaving $(g_n \circ g_{n-1})\#(g_{n-2} \circ g_{n-3})\#\cdots(g_2 \circ g_1)\#$ on the tape. The number of sweeps is just $t(k)$, and hence the total μ_1-time of this phase is $\leq 2t(k) \cdot n = O(n)$.

M copies this output to cells $[((n/2)k \cdot 2^k + 1)\ldots(nk \cdot 2^k)]$ of the storage track, and then repeats the process, until the last phase leaves $h := g_n \circ g_{n-1} \circ \ldots \circ g_2 \circ g_1$ on the tape. Since the length of the input halves after each phase, the total μ_1-time is still $O(n)$. This finishes the word problem.

To solve the trajectory problem, M uses the stored intermediate results to recover the path $(w, g_1(w), g_2(g_1(w)), \ldots, h(w)) =: (w, w_1, w_2, \ldots, w_n)$ of the given $w \in \{0,1\}^k$. Arguing inductively from the base case $(w, h(w))$, we may suppose that M has just finished computing the path $(w, w_2, w_4, \ldots, w_{n-2}, w_n)$. M shuffles this with the string $g_1\#g_3\#g_5\#\ldots\#g_{n-1}$ and then simulates in the above manner a TM T' that given a g and a w computes $g(w)$. All this takes μ_1-time $O(n)$. $\square$

The following presupposes that all BMs M are described in such a way that the alphabet Γ_M of M can be represented by a uniform *code* over $\{0,1\}^*$. This *code* is extended to represent monoids $\mathcal{M}$ as described above.

THEOREM 6.3. *There is a BM M_U and a computable function code such that for any BM M and rational $d \geq 1$, there is a constant K such that for all inputs x to M, M_U on input $(code(M), code(x), d)$ simulates $M(x)$ within μ_d-time $K{\cdot}\mu_d$-time(M, x).*

Proof. M_U uses the alphabet $\Gamma_U := \{0, 1, @, \$, (0, \#), (1, \#), (@, \#), \Delta, B\}$. By Theorem 2.1, we may suppose that M has a single GST $S = (Q, \Gamma_M, \Gamma_M, \delta, \rho, s_0)$. Let $k := \lceil \log_2 |\Gamma_M| \rceil$, and let l be the least integer above $log_2|Q|$ that is a multiple of k. The *code* function on strings codes each $c \in \Gamma_M$ by a 0-1 string of length k, except that the last bit of $code(c)$ is combined with $\#$ and B is coded by $@^{k-1}(@, \#)$.

The monoid $\mathcal{M}$ of transformations of S is encoded by a k-tuple of elements of the form $code(c)code(g_c)$ over all $c \in \Gamma_M$. Here $code(g_c)$ is as described before Lemma 6.2. Dummy states are added to Q so that $code(g_c)$ has length exactly $2l{\cdot}2^l$; then $code(\mathcal{M})$ has length exactly $2^k(k + 2l{\cdot}2^l)$. Let C be the maximum number of symbols written in any transition of M. The code of S includes a string $code(\rho)$ that gives the output for each transition in δ, padded out with @ symbols to length exactly C (i.e., length Ck under *code*). The rest of the code of M lists the mode-change information for each terminal state of S. Finally, the input x to M is represented by the string $code(x)$ of length $|x|2^k$.

M_U has four tracks: one for the main tape of M, one for the code of M, one for simulating passes by M, and one for scratchwork. M_U uses d to compute $e := \mu_d(a)$, and follows just the part of the proof of Theorem 4.13 that locates the cells $a_j := a \pm 2^{j-1}e$, in order to drop \$ characters there. This allows M_U to pull off from its main track in cells $[a \ldots a_j]$ the *code* of the first $m := 2^{j-1}e/4k$ characters of the string x that M reads in the pass being simulated. (If this pass is a put rather than a pull, then $e = 1$ and x is in cells $[1 \ldots 2^{j-1}]$.) Then M_U changes $code(z)$ to

$$z' := (code(z_0))^j \cdot (code(z_1))^j \cdots (code(z_{m-1}))^j,$$

where $m := |z|$ and $j := 2^k(1 + 2(l/k)2^l)$. This can be done in linear μ_1-time by iterating the procedure for *shuffle* in Lemma 6.1(b). Now for each i, $0 \leq i \leq m-1$, the ith segment of z' has the same length as $code(\mathcal{M})$. Next, M uses "recursive doubling" to change $code(\mathcal{M})$ to $(code(\mathcal{M}))^m$. This also takes only $O(m)$ time. Then the strings z' and $(code(\mathcal{M}))^m$ are interlaced on the scratchwork track. A single pass that matches the labels $code(c)$ to segments of z' then pulls out the word $g_z := g_{z_0} \cdot g_{z_1} \cdots g_{z_{m-1}}$.

M evaluates this word by the procedure of Lemma 6.2, yielding the encoded trajectory $s' := (s_0, s_1, \ldots, s_m)$ of S on input z. By a process similar to that of the last paragraph, M_U then aligns s' with $(code(\rho))^m$ and interleaves them, so that a single pass pulls out the output y of the trajectory. Then $code(y)$ is written to the main tape, erasing the symbols Δ used for padding and translating @ to B. The terminal state s_m of the trajectory is matched against the list that gives the mode information for the next pass of M (Lemma 6.1a), and M_U changes its mode and/or current address accordingly.

If the original pass by M cost μ-time $\mu(a) + m + \mu(b)$, then the simulation takes μ-time $\mu(4a) + O(m) + \mu(4b)$. The constant in the '$O(m)$' depends only on M. We have described M_U as though there were no validity restrictions on passes, but Theorems 4.1 and 2.1 convert M_U to a basic BM while keeping the constant overhead on μ_d-time. $\square$

Remarks: This result implies that there is a fixed, finite collection of GSTs that form an efficient "universal chipset." It might be interesting to explore this set in greater detail. The constant on the '$O(m)$' is on the order of $2^{2(l+k)}(l+k)$. We inquire whether there are other representations of finite automata or their monoids that yield notably more efficient off-line simulations than the standard one used here. The universal simulation in Theorem 6.3 does not preserve w or μ_d-*acc* individually because it uses the method of Theorem 4.13 to compensate for its lack of "foreknowledge" about where a given block move by M will exit. The simulation does preserve memory-efficiency, on account of Proposition 5.3. If, however, we suppose that M is already self-delimiting in a way made transparent by *code*, then we obtain constant overheads in both w and μ-*acc*, and the simulation itself becomes independent of μ.

THEOREM 6.4. *There is a BM M_U and a computable function code such that for any memory cost function μ and any self-delimiting BM M, there is a constant K such that for all inputs x to M, M_U on input $x' = (code(M), code(x))$ simulates $M(x)$ with $w(U, x') \leq Kw(M, x)$ and μ-$acc(U, x') \leq K\mu$-$acc(M, x)$.*

Proof. The function *code* is changed so that it encodes the endmarkers of M by strings that begin with '\$'. Then M_U pulls off the portion x of its main track up to \$. The rest of the operation of M_U is the same, and the bounds now require only the tracking property of μ. (If the notion of "self-delimiting" is weakened as discussed before Definition 3.3, then we can have M_U first test whether a GST S exits on the second symbol of x.) ☐

To use these results for diagonalization, we need two preliminary lemmas. Recall that a function t is *μ-time constructible* if $t(n)$ is computable in binary notation in μ-time $O(t(n))$. Since all of n must be read, t must be $\Omega(\log n)$.

LEMMA 6.5. *If a BM M is started on an input of length n, then any pass by M either takes μ_1-time $O(n)$, or else no more than doubles the accumulated μ-time before the pass.*

Proof. Any portion of the tape other than the input that is read in the pass must have been previously written in some other pass or passes. (Technically, this uses our stipulation that B is an endmarker for GSTs.) Thus the conclusion follows. ☐

LEMMA 6.6. *For any memory cost function μ that is μ-time constructible, a BM M can maintain a running total of its own μ-time with only a constant-factor slowdown.*

Proof. To count the number $m = |b - a| + 1$ of transitions made by one of its GST chips S in a given pass, a BM M can invoke a "dummy copy" of S that copies the content x of the cells up to where S exits to a fresh track, and then count $|x|$ on that track by the $O(m)$-time procedure of Example 2.3. Then M invokes S itself and continues operation as normal. Since μ is $\Omega(\log n)$, the current address a can be copied and updated on a separate track in μ-time $O(\mu(a))$. Also in a single pass, M can add a and m in μ-time $O(\mu(a) + m)$, and thus obtain b itself. M then calculates $\mu(b)$ in μ-time $O(\mu(b))$, and finally adds $k := \mu(a) + m + \mu(b)$ to its running total t of μ-time. In case t is much longer than k, we want the work to be proportional to $|k|$, not to $|t|$. Standard "carry-save" techniques, or alternatively an argument that long carries cannot occur too often, suffice for this. ☐

THEOREM 6.7. *Let $d \geq 1$ be rational, and let t_1 and t_2 be functions such that t_2 is μ_d-time constructible, and t_1 is $o(t_2)$. Then* DμTIME$[t_1]$ *is properly contained in* DμTIME$[t_2]$.

Proof. The proof of Theorem 6.3 encoded BMs M over the alphabet Γ_U, but let *code'* re-code M over $(00 \cup 11)^*$. We build a BM M_D that accepts a language $D \in \mathrm{D}\mu\mathrm{TIME}[t_2] \setminus \mathrm{D}\mu\mathrm{TIME}[t_1]$ as follows. M_D has two extra tracks on which it logs its own μ-time, as in Lemma 6.6. On any input x, M_D first calculates $n := |x|$, and then calculates $t_2(n)$ on its "clock track." Next, M_D lets w be the maximal initial segment of doubled bits of x. Since the set $\{\, code(M) : M \text{ is a BM} \,\}$ is recursive, M_D can decide whether w is the *code'* of a BM M in some time $U(n)$. The device of using w ensures that there are ∞-many inputs in which any given BM M is presented to M_D. If w is not a valid code, M_D halts and rejects.

If so, M_D runs M_U on input $code(M) \cdot code(x)$, except that after every pass by M_U, M_D calculates the μ-time of the pass and subtracts it from the total on its clock tape. If the total ever falls below $t_2(n)/2$, M_D halts and rejects. Otherwise, if the simulation of $M(x)$ finishes before the clock "rings," M_D rejects if M accepts, and accepts if M rejects. By Lemma 6.5, the total μ-time of M_D never exceeds $t_2(n)$.

Now let L be accepted by a BM M_L that runs in μ-time $t_1(n)$. Let K_1 be the constant overhead for M_U to simulate M_L in Theorem 6.3, and let K_2 be the overhead in Lemma 6.6. Since t_1 is $o(t_2)$, there exists an x such that $t_2(|x|)/t_1(|x|) > 4K_1K_2$, the maximal initial segment $w \in (00 \cup 11)^*$ of x is $code'(M_L)$, and $U(|w|) < |x|$. Then the simulation of $M_L(x)$ by M_D finishes within μ-time $(1/2)t_2(|x|)$, and $M_D(x) \neq M_L(x)$. $\square$

It is natural to ask whether the classes $\mathrm{D}\mu_d\mathrm{TIME}[t(n)]$ also form a tight hierarchy when t is held constant and d varies. The next section relates this to questions of determinism versus nondeterminism.

We observe finally that the BM in its original, reduced, and buffer forms all give the same definition of $\mathrm{D}\mu_{\log}\mathrm{TIME}[t(n)]$, and we have:

THEOREM 6.8. *For any time functions t_1, t_2 such that $t_1(n) \geq n$, $t_1 = o(t_2)$, and t_2 is $\mu_{\log}$-time constructible,* $\mathrm{D}\mu_{\log}\mathrm{TIME}[t_1]$ *is properly contained in* $\mathrm{D}\mu_{\log}\mathrm{TIME}[t_2]$.

Proof. Here the strict boundary condition is not an issue, but the efficient universal simulation still requires delimiting the read block in advance. The idea is to locate cells $a_1, a_2, a_3 \ldots$ in the proof of Theorem 4.13 without addressing by the following trick. As in Theorem 4.14, the current address a_0 is already stored and $e = |a_0|$. In a rightward pull, rather than add $a_0 + e$, M' puts a_0 itself in binary rightward from cell a_0 on a separate track, appending an endmarker \$. By "recursively doubling" the string a_0, M' can likewise delimit the cells $a_2, a_3, \ldots$ Leftward pull steps are handled similarly, and put steps do not need $\mu_{\log}(a_0)$ at all. This is all that is needed for the efficient universal simulation. The remainder follows as above, since $\mu_{\log}$ is $\mu_{\log}$-time constructible—in fact, $\mu_{\log}(a) = |a|$ is computable in μ_1-time $O(|a|)$. $\square$

A similar statement holds for the perhaps-larger $\mu_{\log}$-time classes for the BM variants that do use addressing.

7. Complexity Theory and the BM Model. Our first result shows that the construction in the *Hennie-Stearns theorem* [33], which states that any multitape TM that runs in time $t(n)$ can be simulated by a 2-tape TM in time $t(n)\log t(n)$, is memory-efficient on the BM under μ_1. It has been observed in general that this construction is an efficient caching strategy. $\mathrm{DTIME}[t(n)]$ refers to TM time, and DLIN stands for $\mathrm{DTIME}[O(n)]$.

THEOREM 7.1. *For any time function t,* $\mathrm{DTIME}[t(n)] \subseteq \mathrm{D}\mu_1\mathrm{TIME}[t(n)\log t(n)]$.

Proof. With reference to the treatment in [36], let M_1 be a multitape TM with alphabet Γ that runs in time $t(n)$, and let M_2 be the two-tape TM in the proof. The k-many tapes of M_1 are simulated on $2k$-many tracks of the first tape of M_2 so that all tape heads of M_1 are maintained on cell 0 of each track. M_2 uses its second tape only to transport blocks of the form $[2^{j-1} \dots 2^j - 1]$ from one part of the first tape to another. The functions used in these moves are homomorphisms between the alphabets Γ^{2k} and Γ^k that pack and unpack characters in blocks. Thus a BM M_3 simulating M_2 can compute each move in a single GST pass. By the structure of the blocks, any pass that incurs a memory-access charge of $\mu_1(2^j) = 2^j$ simulates at least 2^{j-1} moves of M_2. Hence the work and the μ_1 charges to M_3 are both $O(t(n) \log t(n))$. $\square$

We do not know whether the random access capability of a BM can be exploited to give an $O(t \log t)$ simulation that holds the work to $O(t)$, even for $\mu = \mu_{\log}$. Indeed, $O(t \log t)$ is the best bound we know for all memory cost functions μ between $\mu_{\log}$ and μ_1. One consequence of this proposition is that sets in DLIN can be padded out to sets in TLIN.

Corollary 7.2.
(a) For every $L \in$ DLIN, *the language* $\{\, x\#0^{|x|\log|x|} : x \in L \,\}$ *belongs to* TLIN.
(b) TLIN *contains* P*-complete languages, so* TLIN $\subseteq$ NC $\iff$ P = NC. $\square$

Hence it is unlikely that all TLIN functions can be computed in polylog-many passes like the examples in this paper. If a BM quickly compresses the amount of information remaining to be processed into cells $[0 \dots \sqrt{n}]$, it can then spend $O(\sqrt{n})$ time accessing these cells in any order desired and still run in linear μ_1-time.

Theorem 7.3. *Let M be a BM that runs in μ-time $t(n)$ and space $s(n)$. Then we can find a DTM T that simulates M in time $O[t(n)s(n)/\mu(s(n))]$.*

Proof. T has two tapes: one for the main tape of M, and one used as temporary storage for the output in passes. (If M has the buffer mechanism, then the second tape of T simulates the buffer.) Let s stand for $s(n)$. Consider a move by M that changes the current address a to $\lfloor a/2 \rfloor$. T can find this cell in at most $3a/2$ steps by keeping count with its second tape. Since $s/a \geq 1$, the tracking property $\mu(Na) \leq N\mu(a)$ with $N := s/a$ gives $a/\mu(a) \leq s/\mu(s)$. Hence the ratio of the time used by T to the μ-time charged to M stays $O[s/\mu(s)]$. The same holds for the moves $a := 2a$ and $a := 2a+1$. T has every GST S of M in its finite control, and simulates a pull by writing $S[a \dots b]$ to its second tape, moving to cell 0, copying $S[a \dots b]$ over the first tape, and moving back to cell a. Both this and the analogous simulation of a put by T take time $O(a + b)$, and even the ratio of this to the memory access charges $\mu(a) + \mu(b)$, not even counting the number of bits processed by M, keeps the running total of the time logged by T below $t(n)s/\mu(s)$. $\square$

Corollary 7.4. *For any time bound $t(n) \geq n$,* Dμ_1TIME$[t(n)] \subseteq$ DTIME$[t(n)]$. *In particular,* TLIN $\subseteq$ DLIN. $\square$

More generally, for any $d \geq 1$, Dμ_dTIME$[t(n)] \subseteq$ DTIME$[t^{2-(1/d)}(n)]$. Allowing TMs to have d-dimensional tapes brings this back to a linear-time simulation:

Lemma 7.5. *For any integer $d \geq 1$ and time bound $t(n) \geq n$, a BM M that runs in μ_d-time $t(n)$ can be simulated in time $O(t(n))$ by a d-dimensional TM T.*

Proof. T has one d-dimensional tape on which it winds the main tape of M in a spiral about the origin, and one linear tape on which it buffers outputs by the GST

S of M. In any pass that incurs a μ_d charge of $a^{1/d}$, T can walk between cell a and the origin within $a^{1/d}$ steps and complete the move. □

Let us say that a language or function in $\mathrm{D}\mu_d\mathrm{TIME}[O(n)]$ *has dimension* d. For a problem above linear time, we could say that its *dimensionality* is the least d, if any, for which the problem has relatively optimal BM programs that are μ_d-efficient (see Definition 2.10). The main robustness theorem is our justification for this concept of dimensionality. Lemma 7.5 says that it is no less restrictive than the older concept given by d-dimensional Turing machines. For $d > 1$ we suspect that it is noticeably more restrictive. The d-dimensional tape reduction theorem of Paul, Seiferas, and Simon [58] gives $t'(n)$ roughly equal to $t(n)^{1+1/d}$, and when ported to a BM, incurs memory access charges close to $t(n)^{1+2/d}$. Intuitively, the problem is that a d-dimensional TM can change the direction of motion of its tape head(s) at any step, whereas this would be considered a break in pipelining for the simulating BM, and thus subject to a memory-access charge.

We write $\mathrm{RAM\text{-}TIME}^{\log}$ for time on the log-cost RAM. A log-cost RAM can be simulated with constant-factor overhead by a TM with one binary tree-structured tape and one standard worktape [57], and the latter is simulated in real time by a RAM-TM.

PROPOSITION 7.6. *For any time function* t,
(a) $\mathrm{RAM\text{-}TIME}^{\log}[t(n)] \subseteq \mathrm{D}\mu_{\log}\mathrm{TIME}[t(n)\log t(n)]$.
(b) $\mathrm{D}\mu_{\log}\mathrm{TIME}[t(n)] \subseteq \mathrm{RAM\text{-}TIME}^{\log}[t(n)\log t(n)]$.

Proof. Straightforward simulations give these bounds. (The extra $\log t(n)$ factor in (b) dominates a factor of $\log\log n$ that was observed by [44] for the simulation of a TM (or RAM-TM) by a log-cost RAM.) □

For *quasilinear time*, i.e. time $qlin = n(\log n)^{O(1)}$, the extra $\log n$ factors in Theorem 7.1 and Proposition 7.6 do not matter. Following Schnorr [65], we write DQL and NQL for the TM time classes $\mathrm{DTIME}[qlin]$ and $\mathrm{NTIME}[qlin]$. Gurevich and Shelah proved that $\mathrm{RAM\text{-}TIME}^{\log}[qlin]$ is the same as deterministic nearly linear time on the RAM-TM and several other RAM-like models, and perhaps more surprisingly, that the nondeterministic counterparts of these classes are all equal to NQL.

COROLLARY 7.7.
(a) $\mathrm{D}\mu_1\mathrm{TIME}[qlin] = \mathrm{DQL}$.
(b) $\mathrm{D}\mu_{\log}\mathrm{TIME}[qlin] = \mathrm{RAM\text{-}TIME}^{\log}[qlin] \subseteq \mathrm{NQL}$. □

Thus obtaining any separation by more than factors of $\log n$ of the classes $\mathrm{D}\mu\mathrm{TIME}[O(n)]$ as μ varies from μ_1 through μ_d to $\mu_{\log}$ runs into the problem of whether $\mathrm{DQL} \neq \mathrm{NQL}$, which seems as hard as showing $\mathrm{P} \neq \mathrm{NP}$. Whether they can be separated by even one $\log n$ factor is discussed in the next section.

8. Open Problems and Further Research. The following languages have been much studied in connection with linear-time algorithms and nonlinear lower bounds. We suppose that the lists in L_{dup} and L_{int} are all normal.

(a) Pattern matching: $L_{pat} = \{\, p\#t : (\exists u, v \in \{\,0,1\,\}^*)\, t = upv \,\}$.
(b) Element (non)distinctness: $L_{dup} = \{\, x_1\#\ldots\#x_m : (\exists i,j)\, i < j \wedge x_i = x_j \,\}$.
(c) List intersection: $L_{int} = \{\, x_1\#\ldots\#x_m,\ y_1\#\ldots\#y_m : (\exists i,j)\, x_i = y_j \,\}$.
(d) Triangle: $L_\Delta = \{A : A$ is the adjacency matrix of an undirected graph that contains a triangle$\}$.

L_{pat} belongs to DLIN (see [25, 23]), and was recently shown not to be solvable by a one-way non-sensing multihead DFA [42]. L_{dup} and L_{int} can be solved in linear time by a RAM or RAM-TM that treats list elements as cell addresses. L_Δ is not believed to be solvable in linear time on a RAM at all. The best method known involves computing $A^2 + A$, and squaring $n \times n$ integer matrices takes time approximately $N^{1.188}$, where $N = n^2$, by the methods of [19]. (For directed triangles, cubing A is the best way known.)

Open Problem 1. Do any of the above languages belong to TLINΓ If not, prove nonlinear lower bounds.

A BM can be made nondeterministic (NBM) by letting $\delta(q, c)$ be multiply valued, and more strongly, by using nondeterministic GSTs or GSM mappings in block moves. Define NTLIN to be linear time for NBMs of the *weaker* kind. Then all four of the above languages belong to NTLIN. Moreover, they require only $O(\log n)$ bits of nondeterminism.

Open Problem 2. Is NTLIN $\neq$ TLINΓ For reasonable μ and time bounds t, is there a general separation of NμTIME$[t(n)]$ from DμTIME$[t(n)]$Γ

Grandjean [27, 28] shows that a few NP-complete languages are also hard for NLIN under TM linear time reductions, and hence by the theorem of [56] lie outside DLIN, not to mention TLIN. However, these languages seem not to belong to NTLIN, nor even to linear time for NBMs of the stronger kind. The main robustness theorem and subsequent simulations hold for the weaker kind of nondeterminism, but our proofs do not work for the stronger because they re-run the GST S used in a pass. We suspect that different proofs will give similar results. A separation of the two kinds can be shown with regard to the pass count measure $R(n)$, which serves as a measure of parallel time (e.g. $R(n) = \text{polylog}(n)$ and polynomial work $w(n)$ by deterministic BMs characterizes NC [62]). P. van Emde Boas [personal communication, 1994] has observed that while deterministic BMs and NBMs of the weaker kind belong to the *second machine class* of [68] with $R(n)$ as time measure, NBMs of the stronger kind have properties shown there to place models beyond the second machine class. Related to Problem 2 is whether the classes Dμ_dTIME$[O(n)]$ differ as d varies. It is also natural to study *memory-efficient* reductions among problems.

The following idea for obtaining such separations and proving nonlinear lower bounds in μ-time on a deterministic BM M suggests itself: Let $\Gamma_{M,x}$ stand for the set of access points used in the computation of the BM M on input x. In order for M to run in linear μ-time, $\Gamma_{M,x}$ must thin out at the high end of memory. In particular for $\mu = \mu_1$, there are long segments between access points that can be visited only a constant number of times. The technical difficulty is that block moves can still transport information processed in low memory to these segments, and the proof of Theorem 7.1 suggests that a lower bound of $\Omega[n \log n]$ may be the best achievable in this manner. In general, we advance the BM as a logical next step in the longstanding program of proving nonlinear lower bounds for natural models of computation. In particular, we ask whether the techniques used by Dietzfelbinger, Maass, and Schnitger [20] to obtain lower bounds for Boolean matrix transpose and several sorting-related functions on a certain restricted two-tape TM can be applied to the differently-restricted kind of two-tape TM in Theorems 7.1 and 7.3. The latter kind is equivalent to a TM with one worktape and one pushdown store with the restriction that after any Pop, the entire store must be emptied before the next Push.

We have found two variants to the BM model that seem to depart from the cluster of robustness results shown in this paper. They relate to generally-known issues of *delay* in computations. The first definition is the special case for GSTs of Manacher's notion of a "fractional on-line RAM algorithm with steady-paced output" [53].

DEFINITION 8.1. Let $d \geq 0$ and $e \geq 1$ be integers. A GST S runs in *fixed output delay* d/e if for every terminal trajectory $(q_0, x_0, q_1, \ldots, x_{m-1}, q_m)$, and each $i \leq m-2$, $|\rho(q_i, x_i)| = d$ if e divides $i+1$, $= 0$ otherwise. For the exiting transition, $|\rho(q_{m-1}, x_{m-1})|$ depends only on $(m \bmod e)$. The quantity $C := d/e$ is called the *expansion factor* of S.

Note that the case $d = 0$ is allowed. Every GST function g can be written as $e \circ f$, where f is fixed-delay and e is an erasing homomorphism: pad each output of the GST for g to the same length with '@' symbols, and let e erase them. A k-input GST *with stationary moves allowed* may keep any of its input heads stationary in a transition. Such a machine can be converted to an equivalent form coded like an ordinary GST in which every state q has a label $j \in \{1, \ldots, k\}$ such that q reads and advances only the head on tape j.

DEFINITION 8.2. (a) A BM *runs in fixed output delay* if every GST chip in M runs in fixed output delay.
(b) A *pause buffer BM* is a BM with buffer whose put steps may use 2-input GSTs with variable input delay (cf. Proposition 4.3).

Put another way, the BM model presented in this paper requires fixed delay in reading input but not in writing output, while (a) requires both and (b) requires neither. We did not adopt (b) because we feel that stationary moves by a 2-GST in the course of a pass require communication between the heads, insofar as further movements depend on the current input symbols, and hence should incur memory-access charges. We defend our choice against a similar criticism that would require (a) by contending that in a single-tape GST pass, the motion of the read head is not affected by the write head, and the motion of the write head depends only on local factors as bits come in to it. Also, every BM has a limit C on the number of output bits per input bit read by a GST. The main robustness theorem, in particular the ability to forecast the length of the output of a pass by fixed-delay means shown in Theorem 4.13, satisfy our doubts about this.

The robustness results in this paper do carry over to the case of fixed output delay:

THEOREM 8.1. *For any rational $d \geq 1$, the fixed-delay restrictions of the BM and all the variants defined in Section 3 simulate each other up to constant factors in μ_d-time.*

Proof. All auxiliary operations in the simulations in Section 4 use GSTs that run in fixed output delay, except for the second, unpadded run of the GST S in Theorem 4.13. However, if S already runs in fixed output delay, so does this run. □

Under the proof of Theorem 2.1 the corresponding notion for the reduced form of the model is "fixed delay after the initial transition." Our proof of efficient universal simulation does not quite carry over for fixed output delay because the quantities k and l in the proof of theorem 6.3 may differ for different M. The operations that pull off the word g_z and the padded output $code(y)$ run in "stride" a function of k and l, but this is not fixed. We believe that the proof can be modified to do so under a different representation scheme for monoids.

Whether a similar robustness theorem holds for the pause-buffer BM leads to an open problem of independent interest: can every k-input GST be simulated by a composition tree of 2-input GSTs when stationary moves are allowedΓ The questions of the power of both variants versus the basic BM can be put in concrete terms.

Open Problem 3. Can the homomorphism $Er_2 : \{0,1,2\}^* \to \{0,1\}^*$, which erases all 2's in its argument, be computed in linear μ_1-time by a BM that runs in fixed output delayΓ

Open Problem 4. For every 2-input GST S with stationary moves allowed, does the function $S'(x\#y) := S(x,y)$ belong to TLINΓ

Theorem 8.2.

(a) *The answer to Problem 3 is 'yes' iff for every memory cost function μ and BM M, there is a BM M' that runs in fixed output delay and simulates M linearly under μ.*

(b) *The answer to Problem 4 is 'yes' iff for every memory cost function μ and pause-buffer BM M, there is a BM M' that simulates M linearly under μ.*

Proof. For the forward implication of (a), M' pads every output by M with @ symbols, coding the rest over $\{0,1\}^*$, and runs $Er_@$ on a separate track to remove the padding. That of (b) is proved along the lines of Proposition 4.3. The reverse implications are immediate, and all this needs only the tracking property of μ. □

Alon and Maass [4] prove substantial time-space tradeoffs for the related "sequence equality" problem SE[n]: given $x, y \in \{0,1,2\}^n$, does $Er_2(x) = Er_2(y)$Γ We inquire whether their techniques, or those of [54], can be adapted to the BM. The BM in Theorem 7.1 runs in output delay 1/2, 1, or 2 for all passes, so the two kinds of BM can be separated by no more than a log factor. A related question is whether every language in TLIN, with or without fixed output delay, has linear-sized circuits.

Further avenues for research include analyzing implementations of certain important algorithms on the BM, as done for the BT and UMH in [2, 5]. Here the BM is helped by its proximity to the Pratt-Stockmeyer vector machine, since conserving memory-access charges and parallel time often lead to similar methods. One can also study storage that is expressly laid out on a 2D grid or in 3D space, where a *pass* might be defined to follow either a 1D line or a 2D plane. We expect the former not to be much different from the BM model with its 1D tape, and we also note that CD-ROM and several current 2D drive technologies use long 1D tracks. The important issue may not be so much the topology of the memory itself, but whether "locality is one-dimensional" for purposes of pipelining.

Last, we ask about meaningful technical improvements to the simulations in this paper. The lone obstacle to extending the main robustness theorem for $\mu = \mu_{\log}$ is the simulation of random access by tree access in Lemma 4.6. The constants on our universal simulation are fairly large, and we seek a more-efficient way of representing monoids and computing the products. Two more questions are whether the BM loses power if the move option $a := 2a+1$ is eliminated, and whether the number m of markers in a finger BM can be reduced to $m-1$ or to 4 without multiplying the number of block moves by a factor of $\log t(n)$.

9. Conclusion. In common with motivations expressed in [2] and [5], the BM model fosters a finer analysis of many theoretical algorithms in terms of how they use memory, and how they really behave in running time when certain practicalities of implementation are taken into account. We have shown that the BM model is quite robust, and that the concept of functions and languages being computable in a memory-efficient manner does not depend on technical details of setting up the model. The richer forms of the model are fairly natural to program, providing random access and the convenience of regarding finite transductions such as addition and vector Booleans as basic operations. The tightest form of the model is syntactically simple, retains the bit-string concreteness of the TM, and seems to be a tractable object of study for lower bound arguments. The robustness is evidence that our abstraction is "right."

In contrast to the extensive study of polynomial-time computation, very little is known about linear time computation. Owing to an apparent lack of linear-time robustness among various kinds of TMs, RAMs, and other machines, several authorities have queried their suitability as a model for computation in $O(n)$ time. Since we have μ as a parameter we have admittedly not given a single answer to the question "What is Linear TimeΓ", and leave TLIN, Dμ_2TIME[$O(n)$], and Dμ_3TIME[$O(n)$] as leading candidates. However, the BM model does supply a robust yardstick for assessing the complexity of many natural combinatorial problems, and for investigating the structure of several other linear-time complexity classes. It has a tight deterministic time hierarchy right down to linear time. The efficient universal simulator which we have constructed to show this result uses the word problem for finite monoids in an interesting manner. The longstanding program of showing nonlinear lower bounds in reasonable models of computation has progressed up to machines apparently just below the BM (under μ_1) in power, so that attacking the problems given here seems a logical next step. The authors of [3] refer to the "challenging open problem" of extending their results when bit-manipulations for dissecting records are available. The bit operations given to the BM seem to be an appropriate setting for this problem. A true measure of the usefulness of the BM model will be whether it provides good ground for developing and connecting methods that solve older problems not framed with the term "BM." We offer the technical content of this paper as appropriately diligent spadework.

Acknowledgments.. I would like to thank Professor Michael C. Loui for comments on preliminary drafts of this work, and for bringing [2] and [5] to my attention. Professors Pierre McKenzie and Gilles Brassard gave me a helpful early opportunity to test these ideas at an open forum in a University of Montreal seminar. Special thanks are due to Professors Klaus Ambos-Spies, Steven Homer, Uwe Schöning, and other organizers of the 1992 Schloss Dagstuhl Workshop on Structural Complexity for inviting me to give a presentation out of which the present work grew. Last, I thank an anonymous referee and several colleagues for helpful suggestions on nomenclature and presentation.

REFERENCES

[1] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir, *A model for hierarchical memory*, in Proc. 19th Annual ACM Symposium on the Theory of Computing, 1987, pp. 305–314.

[2] A. Aggarwal, A. Chandra, and M. Snir, *Hierarchical memory with block transfer*, in Proc. 28th Annual IEEE Symposium on Foundations of Computer Science, 1987, pp. 204–216.

[3] A. Aggarwal and J. Vitter, *The input-output complexity of sorting and related problems*, Comm. ACM, 31 (1988), pp. 1116–1127.

[4] N. Alon and W. Maass, *Meanders and their application to lower bound arguments*, J. Comp. Sys. Sci., 37 (1988), pp. 118–129.

[5] B. Alpern, L. Carter, and E. Feig, *Uniform memory hierarchies*, in Proc. 31st Annual IEEE Symposium on Foundations of Computer Science, 1990, pp. 600–608.

[6] J. Balcázar, J. Díaz, and J. Gabarró, *Structural Complexity Theory*, Springer Verlag, 1988.

[7] D. M. Barrington, N. Immerman, and H. Straubing, *On uniformity within* NC^1, in Proc. 3rd Annual IEEE Conference on Structure in Complexity Theory, 1988, pp. 47–59.

[8] ———, *On uniformity within* NC^1, J. Comp. Sys. Sci., 41 (1990), pp. 274–306.

[9] A. Ben-Amram and Z. Galil, *Lower bounds for data structure problems on RAMs*, in Proc. 32nd Annual IEEE Symposium on Foundations of Computer Science, 1991, pp. 622–631.

[10] ———, *On pointers versus addresses*, J. ACM, 39 (1992), pp. 617–648.

[11] G. Blelloch, *Vector Models for Data-Parallel Computing*, MIT Press, 1990.

[12] M. Blum, *A machine-independent theory of the complexity of recursive functions*, J. ACM, 14 (1967), pp. 322–336.

[13] R. Book and S. Greibach, *Quasi-realtime languages*, Math. Sys. Thy., 4 (1970), pp. 97–111.

[14] A. Chandra, D. Kozen, and L. Stockmeyer, *Alternation*, J. ACM, 28 (1981), pp. 114–133.

[15] J. Chang, O. Ibarra, and A. Vergis, *On the power of one-way communication*, J. ACM, 35 (1988), pp. 697–726.

[16] J. Chen and C. Yap, *Reversal complexity*, SIAM J. Comput., 20 (1991), pp. 622–638.

[17] M. Conner, *Sequential machines realized by group representations*, Info. Comp., 85 (1990), pp. 183–201.

[18] S. Cook and R. Reckhow, *Time bounded random access machines*, J. Comp. Sys. Sci., 7 (1973), pp. 354–375.

[19] D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetical progressions*, J. Symbolic Computation, 9 (1990), pp. 251–280.

[20] M. Dietzfelbinger, W. Maass, and G. Schnitger, *The complexity of matrix transposition on one-tape off-line Turing machines*, Theor. Comp. Sci., 82 (1991), pp. 113–129.

[21] C. Elgot and A. Robinson, *Random-access stored-program machines*, J. ACM, 11 (1964), pp. 365–399.

[22] Y. Feldman and E. Shapiro, *Spatial machines: A more-realistic approach to parallel computation*, Comm. ACM, 35 (1992), pp. 60–73.

[23] P. Fischer, A. Meyer, and A. Rosenberg, *Real-time simulations of multihead tape units*, J. ACM, 19 (1972), pp. 590–607.

[24] M. Fürer, *Data structures for distributed counting*, J. Comp. Sys. Sci., 29 (1984), pp. 231–243.

[25] Z. Galil and J. Seiferas, *Time-space optimal string matching*, J. Comp. Sys. Sci., 26 (1983), pp. 280–294.

[26] E. Graedel, *On the notion of linear-time computability*, International Journal of Foundations of Computer Science, 1 (1990), pp. 295–307.

[27] E. Grandjean, *A natural NP-complete problem with a nontrivial lower bound*, SIAM J. Comput., 17 (1988), pp. 786–809.

[28] ———, *A nontrivial lower bound for an NP problem on automata*, SIAM J. Comput., 19 (1990), pp. 438–451.

[29] E. Grandjean and J. Robson, *RAM with compact memory: a robust and realistic model of computation*, in CSL '90: Proceedings of the 4th Annual Workshop in Computer Science Logic, vol. 533 of Lect. Notes in Comp. Sci., Springer Verlag, 1991, pp. 195–233.

[30] Y. Gurevich and S. Shelah, *Nearly-linear time*, in Proceedings, Logic at Botik '89, vol. 363 of Lect. Notes in Comp. Sci., Springer Verlag, 1989, pp. 108–118.

[31] J. Hartmanis and R. Stearns, *On the computational complexity of algorithms*, Transactions of the AMS, 117 (1965), pp. 285–306.

[32] ———, *Algebraic Structure Theory of Sequential Machines*, Prentice-Hall, Englewood Cliffs, NJ, 1966.

[33] F. Hennie and R. Stearns, *Two–way simulation of multitape Turing machines*, J. ACM, 13 (1966), pp. 533–546.

[34] T. Heywood and S. Ranka, *A practical hierarchical model of parallel computation I: The model*, J. Par. Dist. Comp., 16 (1992), pp. 212–232.

[35] J.-W. Hong, *On similarity and duality of computation I*, Info. Comp., 62 (1985), pp. 109–128.

[36] J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison–Wesley, Reading, MA, 1979.

[37] O. Ibarra, *Systolic arrays: characterizations and complexity*, in The Proceedings of the 1986 Conference on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science No. 233, Springer Verlag, 1986, pp. 140–153.

[38] O. Ibarra and T. Jiang, *On one-way cellular arrays*, SIAM J. Comput., 16 (1987), pp. 1135–1153.

[39] O. Ibarra and S. Kim, *Characterizations and computational complexity of systolic trellis automata*, Theor. Comp. Sci., 29 (1984), pp. 123–153.

[40] O. Ibarra, S. Kim, and M. Palis, *Designing systolic algorithms using sequential machines*, IEEE Transactions on Computing, 35 (1986), pp. 531–542.

[41] O. Ibarra, M. Palis, and S. Kim, *Some results concerning linear iterative (systolic) arrays*, J. Par. Dist. Comp., 2 (1985), pp. 182–218.

[42] T. Jiang and M. Li, *K one-way heads cannot do string-matching*, in Proc. 25th Annual ACM Symposium on the Theory of Computing, 1993, pp. 62–70.

[43] T. Kameda and R. Vollmar, *Note on tape reversal complexity of languages*, Info. Control, 17 (1970), pp. 203–215.

[44] J. Katajainen, J. van Leeuwen, and M. Penttonen, *Fast simulation of Turing machines by random access machines*, SIAM J. Comput., 17 (1988), pp. 77–88.

[45] A. Kolmogorov and V. Uspenskii, *On the definition of an algorithm*, Uspekhi Mat. Nauk, 13 (1958), pp. 3–28. English transl. in *Russian Math Surveys* **30** (1963) 217–245.

[46] R. Kosaraju, *Real-time simulation of concatenable double-ended queues by double-ended queues*, in Proc. 11th Annual ACM Symposium on the Theory of Computing, 1979, pp. 346–351.

[47] K. Krohn and J. Rhodes, *Algebraic theory of machines. I. prime decomposition theorem for finite semigroups and machines*, Trans. AMS, 116 (1965), pp. 450–464.

[48] K. Krohn, J. Rhodes, and B. Tilson, *The prime decomposition theorem of the algebraic theory of machines*, in Algebraic Theory of Machines, Languages, and Semigroups, M. Arbib, ed., Academic Press, 1968. Ch. 5; see also chs. 4 and 6–9.

[49] D. Leivant, *Descriptive characterizations of computational complexity*, J. Comp. Sys. Sci., 39 (1989), pp. 51–83.

[50] M. Loui, *Simulations among multidimensional Turing machines*, Theor. Comp. Sci., 21 (1981), pp. 145–161.

[51] ———, *Optimal dynamic embedding of trees into arrays*, SIAM J. Comput., 12 (1983), pp. 463–472.

[52] ———, *Minimizing access pointers into trees and arrays*, J. Comp. Sys. Sci., 28 (1984), pp. 359–378.

[53] G. Manacher, *Steady-paced-output and fractional-on-line algorithms on a RAM*, Inf. Proc. Lett., 15 (1982), pp. 47–52.

[54] Y. Mansour, N. Nisan, and P. Tiwari, *The computational complexity of universal hashing*, Theor. Comp. Sci., 107 (1993), pp. 121–133.

[55] D. Muller and F. Preparata, *Bounds to complexities of networks for sorting and switching*, J. ACM, 22 (1975), pp. 195–201.

[56] W. Paul, N. Pippenger, E. Szemerédi, and W. Trotter, *On determinism versus nondeterminism and related problems*, in Proc. 24th Annual IEEE Symposium on Foundations of Computer Science, 1983, pp. 429–438.

[57] W. Paul and R. Reischuk, *On time versus space II*, J. Comp. Sys. Sci., 22 (1981), pp. 312–327.

[58] W. Paul, J. Seiferas, and J. Simon, *An information-theoretic approach to time bounds for on-line computation*, J. Comp. Sys. Sci., 23 (1981), pp. 108–126.

[59] N. Pippenger, *On simultaneous resource bounds*, in Proc. 20th Annual IEEE Symposium on Foundations of Computer Science, 1979, pp. 307–311.

[60] N. Pippenger and M. Fischer, *Relations among complexity measures*, J. ACM, 26 (1979), pp. 361–381.

[61] V. Pratt and L. Stockmeyer, *A characterization of the power of vector machines*, J. Comp. Sys. Sci., 12 (1976), pp. 198–221.

[62] K. Regan, *A new parallel vector model, with exact characterizations of* NC^k, in Proc. 11th Annual Symposium on Theoretical Aspects of Computer Science, vol. 778 of Lect. Notes in Comp. Sci., Springer Verlag, 1994, pp. 289–300.

[63] R. Reischuk, *A fast implementation of multidimensional storage into a tree storage*, Theor. Comp. Sci., 19 (1982), pp. 253–266.

[64] W. Ruzzo, *On uniform circuit complexity*, J. Comp. Sys. Sci., 22 (1981), pp. 365–383.

[65] C. Schnorr, *Satisfiability is quasilinear complete in NQL*, J. ACM, 25 (1978), pp. 136–145.

[66] A. Schönhage, *Storage modification machines*, SIAM J. Comput., 9 (1980), pp. 490–508.

[67] ———, *A nonlinear lower bound for random-access machines under logarithmic cost*, J. ACM,

35 (1988), pp. 748–754.
[68] P. van Emde Boas, *Machine models and simulations,* in Handbook of Theoretical Computer Science, J. V. Leeuwen, ed., Elsevier and MIT Press, 1990, pp. 1–66.
[69] D. Willard, *A density control algorithm for doing insertions and deletions in a sequentially ordered file in a good worst-case time,* Info. Comp., 97 (1992), pp. 150–204.

Appendix: Proof of Theorem 2.1.

For every move state q in M we add a new GST S_q that performs a 1-bit empty pull just to read the currently-scanned character d, and then sends control to $\delta(q,d)$. This modification no more than doubles the μ-access charges, and gives M the following property: for any pass by a GST S_i, the next GST S_k to be invoked (or Halt) is a function only of i and the character c that caused S_i to exit, and there is at most one intervening move. Henceforth we assume that M has this form, and number its GST chips by $S_0,\ldots,S_r$, with S_0 as start chip.

M' uses an alphabet Γ' which includes the alphabet Γ of M, a surrogate blank @, tokens $\{\, s_0,\ldots,s_r \,\}$ for the chips of M, markers $\{\, m_U, m_L, m_R, m_{no}, m_H \,\}$ for the three kinds of move, "no move," and Halt, special *instruction markers* $\{\, I_0,\ldots,I_{12} \,\}$, *plus* certain tuples of length up to 7 of the foregoing characters. We also use @ to indicate that the symbol written to cell 0 is immaterial.

During the simulation, the first component of every tuple in a cell i is the character $c_i \in \Gamma$ in that cell of the tape of M. Except initially, cell 1 holds both c_0 and c_1, so that cell 0 can be overwritten by other characters. This also allows M' to simulate all moves by M without ever moving its own cell-a head back to cell 0. The markers I_0 and I_1 tell M' when the cell-a head of M is in cell 0 or 1. For $a \geq 2$, the heads of M and M' coincide. The other main invariant of the simulation is that the only cell besides cells 0 and 1 to contain multiple symbols is cell a. The two initial moves of M' set up these invariants.

Character(s) read	Action (Initial mode is Ra, $a = 0$.)
c_0, c_1	Pull $[c_0, c_1]$ to cell 0, $a := a$, $mode := 0R$.
$[c_0, c_1], c_1$	Put @ into cell 0 and $[c_1, c_0, s_0, I_0]$ into cell 1, $a := 2a+1$, $mode := Ra$.

The first move must automatically be executed every time M' moves its tape head to a new cell a, $a \geq 2$, since this cell and cell $a+1$ will always contain single characters over Γ. However, the second move is unique to the initialization because cell 1 will never again hold a single character. The cell-a head of M' is now on cell 1, but the I_0 enables M to record that the cell-a head of M is still on cell 0.

The lone GST S of M' includes two copies of each GST S_i of M. The first is a "dummy copy" which simulates S_i but suppresses output until it picks up the character c that causes S_i to exit. On this exiting transition, the dummy outputs a token s_k for the next GST S_k and a token m for the intervening move, or m_{no} for none, or m_H for Halt. The other copy simulates the actual pass by S_i. It has special states that distinguish whether S_i has written zero, one, or at least two output symbols in the pass, since the first one or two symbols of the output y are altered. If S_i performs a pull and $|y| \geq 2$, we define $c'_0 := y_0$ if $y_0 \neq B$, but $c'_0 := c_0$ if $y_0 = B$. Similarly $c'_1 := y_1$ if $y_1 \neq B$, but $c'_1 := c_1$ if $y_1 = B$. On the tape of M', the output y looks like $[c'_0, c'_1, \ldots][c'_1, c'_0, \ldots]y_2 \cdots y_l$, where $l = |y|$. For $|y| \leq 1$, treat the missing y_1 and/or y_0 as B. Besides these functional conventions on s_k, m, c'_0, and c'_1, we omit reference to the address a if it is not changed, and omit the second character read by S when it does not affect control at the large initial branch. Let S_i be the current GST of M.

Character(s) read	Action (Current mode is Ra, $a = 1$.)
$[c_1, c_0, s_i, I_0]$	By the validity conditions (Definition 2.3), the output y by S_i has length at most 2. Hence the next-move token m and next-GST token s_k can be picked up and the output y written in one pass, without needing the dummy copy of S_i. If $m = m_H$, S pulls @ to cell 0 and $[c_1', c_0', I_{12}]$ to cell 1. If $m = m_R$, pulls $@[c_1', c_0', s_k, I_1]$ to signify that the cell-a head of M is now on cell 1. Else S pulls $@[c_1', c_0', s_k, I_0]$, and this step repeats. In each case, *mode* := Ra.
$[c_1, c_0, I_{12}]$	Pull c_0 into cell 0, c_1 into cell 1, and HALT.
$[c_1, c_0, s_i, I_1]$	Simulate S_i as for $[c_1, c_0, s_i, I_0]$ to get m, s_k, and y, but treat c_1 as the first input character to S_i. If $m = m_H$ pull $@[c_1', c_0', I_{12}]$, if $m = m_U$ pull $@[c_1', c_0', s_k, I_0]$, and if $m = m_{no}$, pull $@[c_1', c_0', s_k, I_1]$. In these three cases, the address of M' stays at 1. If $m = m_L$, then pull $@[c_1', c_0', s_k]$ and effect $a := 2a$. If $m = m_R$, pull $@[c_1', c_0', s_k]$ and effect $a := 2a + 1$. In every case, the mode stays Ra.

The last two cases give $a \geq 2$. When $a \geq 2$, the next pass by S encounters a single character $c_a \in \Gamma$ on its start transition (possibly $c_a = B$), and S must perform the first operation above. This overwrites the @ in cell 0. However, the new character $[c_1', c_0', s_k]$ in cell 1 prevents the initial sequence from recurring, viz.:

Character(s) read	Action (Current mode is Ra, $a \geq 2$.)
c_a, c_{a+1}	Pull $[c_a, c_{a+1}]$ to cell 0, $a := a$, *mode* := $0R$.
B	Pull $[@, @, I_0]$ to cell 0, $a := a$, *mode* := $0R$.
$[c_a, c_{a+1}], [c_1, c_0, s_i]$	Put $[c_a, c_0, c_1, s_i, I_2]$ into cell a. If the label of S_i is La then *mode* := La, else *mode* := Ra.
$[c_a, c_0, c_1, s_i, I_2]$	If S_i is labeled $0L$ or $0R$, then pull $[c_0, c_1, c_a, s_i, I_6]$ into cell 0, and *mode* := the mode of S_i. Else S simulates the dummy copy of S_i to find m and s_k, treating c_a as the first input character to S_i, and pulls $[c_0, c_1, c_a, m, s_k, s_i, I_3]$ to cell 0 with *mode* := $0R$.
$[c_0, c_1, c_a, m, s_k, s_i, I_3]$	Put $[c_a, c_0, c_1, m, s_k, s_i, I_4]$ into cell a, *mode* := the mode of S_i.
$[c_a, c_0, c_1, m, s_k, s_i, I_4]$	Simulate the pull by S_i, copying the output y as $[c_0', c_1', c_a, m, I_5][c_1', c_0', s_k]y_2 \cdots y_l$, and change *mode* to $0R$. *Remark:* For c_a to be correct, it is vital that cell a not be overwritten in this pull.
$[c_0, c_1, c_a, m, I_5]$	Put c_a into cell a. On exit, if $m = m_{no}$ then leave a unchanged, if $m = m_U$ effect $a := \lfloor a/2 \rfloor$, if $m = m_L$ effect $a := 2a$, and if $m = m_R$ effect $a := 2a + 1$. In each of these four cases, *mode* := Ra. For $m = m_H$, see below.

If the last move was *up*, i.e. a to $\lfloor a/2 \rfloor$, we may now have $a = 1$ again. Since the "sentinel" in cell 1 is always correctly updated to the next GST S_i, this is handled by:

$[c_1, c_0, s_i]$	Same as for $[c_1, c_0, s_i, I_1]$.

If still $a \geq 2$, then S once again senses single characters in cells a and $a + 1$, and the cycle repeats. The other branch with instruction 6 goes:

$[c_0, c_1, c_a, s_i, I_6]$	Here S_i is labeled $0L$ or $0R$, and this is the current mode. S treats c_0, c_1 as the first two input characters in simulating the dummy copy of S_i, and puts $[c_a, c_0, c_1, m, s_k, s_i, I_7]$ into cell a with *mode* $:= Ra$.
$[c_a, c_0, c_1, m, s_k, s_i, I_7]$	Pull $[c_0, c_1, c_a, m, s_k, s_i, I_8]$ into cell 0, *mode* := the mode of S_i.
$[c_0, c_1, c_a, m, s_k, s_i, I_8]$	Simulate the put by S_i. If the output y is empty or begins with B, let $c'_a := c_a$. Else let $c'_a := y_0$. Copy y as $[c'_a, c_0, c_1, m, s_k, I_9]$, and set *mode* $:= 0R$.
$[c_a, c_0, c_1, m, s_k, I_9]$	Pull $[c_0, c_1, c_a, m, I_5]$ to cell 0 and $[c_1, c_0, s_k]$ to cell 1, *mode* $:= Ra$.

The validity conditions prevent cell a from being overwritten in a pull. It is possible for cell 1 to be overwritten by a leftward put that exits after just one input bit, but this can only happen if $a \leq C$, where C is the maximum number of bits a leftward pull chip of M can write in its first transition. The problem can be solved either by exploiting the ability of M itself to remember C-many characters in its finite control, or by reprogramming M so that no leftward pull chip outputs more than one symbol on its first step. Details are left to the reader.

The final halting routine involves a "staircase" to leave the tape exactly the same as that of M at the end of the computation. It picks up in the case $[c_0, c_1, c_a, m, I_5]$ with $m = m_H$.

$[c_0, c_1, c_a, m_H, I_5]$	Put $[c_a, c_0, c_1, I_{10}]$ into cell a, *mode* $:= Ra$.
$[c_a, c_0, c_1, I_{10}]$	Pull $[c_0, c_1, c_a, I_{11}]$ to cell 0 and $[c_1, c_0, I_{12}]$ to cell 1, with *mode* $:= 0R$.
$[c_0, c_1, c_a, I_{11}]$	Put c_a into cell a, effect $a := \lfloor a/2 \rfloor$, *mode* $:= Ra$.
c_a, c_{a+1}	Pull $[c_a, c_{a+1}]$ to cell 0, *mode* $:= 0R$.
$[c_a, c_{a+1}], [c_1, c_0, I_{12}]$	Put c_a into cell a, effect $a := \lfloor a/2 \rfloor$, *mode* $:= Ra$.
$[c_1, c_0, I_{12}]$	As above, pull c_0 into cell 0, c_1 into cell 1, and HALT.

M' uses exactly the same tape cells as M, making at most eight passes of equal or less cost for each pass by M. The final "staircase" down from cell a is accounted against the μ-charges for M to have moved out to cell a. Hence both the number of bits processed by M' and the μ-*acc* charges to M' are within a constant factor of their counterparts in M.

For the converse simulation of the reduced form S by a BM M, the only technical difficulty is that S may have different exiting transitions on the same character c. The solution is to run a dummy copy of S that outputs a token t for the state in which S exits. Then t is used to send control to the move state of M' that corresponds to the label $l_1(t)$, and thence to a copy of S with the pass-type label $l_2(t)$. The details of running the dummy copy are the same as above. □

By using more "instruction markers" one can make the mode of M' always follow the cycle $Ra, 0R, La, 0L$. Hence the only decision that need depend on the terminal state of the lone GST S is the next origin cell a.