Counting Is Easy

JOEL I. SEIFERAS

University of Rochester, Rochester, New York

AND

PAUL M. B. VITÁNYI

Centre for Mathematics and Computer Science, Amsterdam, The Netherlands

Abstract. For any fixed k, a remarkably simple single-tape Turing machine can simulate k independent counters in real time.

Categories and Subject Descriptors: E.2 [**Data**]: Data Storage Representations—*contiguous representations*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*relations among models*; *bounded-action devices*; F.2.2. [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*sequencing and scheduling*; F.2.3 [**Analysis of Algorithms and Problem Complexity**]: Trade-offs among Complexity Measures; G.2.1 [**Discrete Mathematics**]: Combinatorics—*combinatorial algorithms*

General Terms: Algorithms, Design, Theory, Verification

Additional Key Words and Phrases: Abstract storage unit, counter, counter machine, multicounter machine, oblivious simulation, one-tape Turing machine, on-line simulation, real-time simulation, recursion elimination, redundant number representation, signed-digit number representation, simulation between models

1. *Introduction*

In this paper we describe a remarkably simple real-time simulation, based on just five simple rewriting rules, of any fixed number k of independent counters. On a Turing machine with a single binary work tape, the simulation runs in real time, handling an arbitrary counter command at each step. The space used by the simulation can be held to $(k + \epsilon)\log_2 n$ bits for the first n commands, for any specified $\epsilon > 0$. Consequences and applications are discussed in [10]-[11], where the first single-tape, real-time simulation of multiple counters was reported.

Informally, a *counter* is a storage unit that maintains a single integer (initially 0), incrementing it, decrementing it, or reporting its sign (positive, negative, or zero) on command. Any automaton that responds to each successive command as a counter would is said to *simulate* a counter. (Only for a sign inquiry is the response

The work of J. I. Seiferas was supported in part by the National Science Foundation under grant MCS 81-10430.

Authors' addresses: J. I. Seiferas, Computer Science Department, University of Rochester, Rochester, N.Y. 14627; P. M. B. Vitányi; Centre for Mathematics and Computer Science, P.O. Box 4079, 1009 AB Amsterdam, The Netherlands.

of interest, of course. And zeroness is the only real issue, since a simulator can readily use zero detection to keep track of positivity and negativity in finite-state control.) To simulate k independent counters, an automaton must respond to $3k$ commands: "increment counter number i," "decrement counter number i," and "report the sign of counter number i" ($1 \leq i \leq k$). If there is some fixed bound on the time needed by a simulator to respond to the successive commands it receives, then it simulates in *real time.*

Our real-time k-counter simulator will be a *single-tape Turing machine.* Such an automaton consists of a finite-state control unit with read–write access to an infinite but initially blank binary storage tape (0 in every bit position). Each next step is determined by the current control state, the bit currently scanned by the read–write head on the storage tape, and the most recently received input symbol (in our case, the last command not yet responded to). Each step can involve any of the following actions: A change to the bit scanned by the head on the storage tape, a shift left or right by that head to an adjacent bit position, emission of an output symbol (in our case, a command response), and a state transition by the finite-state control unit.

An apparently stronger notion of real-time simulation would require response to each successive command just *one* step after submission. In the special case of *counter* simulation, however, *any* real-time simulation actually does also yield a real-time simulation in which the command–response delay is just 1. (It is well known that a larger delay can be "swept under the rug" by increasing the size of the alphabet used on the storage tape, but that is not necessary in our case.)

PROPOSITION. *If a single-tape Turing machine can simulate k counters in real time with command–response delay bound d, then a similar single-tape Turing machine (still with only binary tape alphabet) can do so with delay bound* 1.

PROOF. The rough idea is for the delay-1 simulation to use a delay-d simulation to store an appropriate *fraction* of the contents of each of its counters and to maintain all the remainders in finite-state control.

More accurately and precisely, the delay-1 simulation can operate in "phases" of $2kd$ steps, maintaining the following invariant from phase to phase, for the absolute value $|c|$ of each count c:

$$|c| = c_0 + c_1(2kd),$$

where either

$$c_1 > 0 \qquad \text{and} \qquad 2kd \leq c_0 \leq 8kd$$

or

$$c_1 = 0 \qquad \text{and} \qquad 0 \leq c_0 \leq 8kd,$$

and where c_0 and the signs of c and c_1 are stored in finite-state control, and c_1 is stored in the corresponding counter of the delay-d simulation. The $2kd$ commands received in each phase can be handled within finite-state control, increasing or decreasing each c_0 by at most $2kd$. Meanwhile, the $2kd$ steps are enough for one increment or decrement of and one interrogation of each c_1. In each case the simulation should *increment* c_1, as part of a "carry" from c_0, if $c_0 > 6kd$ held when the phase began; and it should *decrement* c_1, as part of a "borrow" for c_0, if $c_0 < 4kd$ held when the phase began, unless c_1 was already zero. For each count, if c_1 was positive when the phase began, then $2kd \leq c_0 \leq 8kd$ will hold when it ends. If c_1 was zero when the phase began, however, c_0 might "underflow" almost to $-2kd$; but, in that case, c_1 will remain zero, so that a sign change in finite-state control will suffice to restore the invariant. Finally, note that there will always be enough information in finite-state control to determine whether a count is currently zero: Each count will be zero just when its c_0 is zero and its c_1 was zero when the current phase began. □

Prior to the breakthrough in [10] and [11], there were at least three weaker simulations in the literature. M. Fischer and Rosenberg [4] showed that the simulation is possible in the case in which only *simultaneous* zeroness of the k counters has to be reported. P. Fischer, et al. [5] showed that a full simulation is possible in cumulative *linear* time (i.e., with *average* delay bounded by a constant, but with no fixed bound on the delay for each individual command). A while later the latter authors showed that *four* Turing machine tapes are as efficient as k counters for *sequence generation* [6]. Fürer's full linear-time simulation [7] requires more than one tape, but two suffice *even if they are otherwise occupied.*

2. *A Peek at an Oblivious Solution*

Using a straightforward unary (or "tally") notation, an automaton with just one storage tape (i.e., a single-tape Turing machine) obviously can simulate a single counter in real time. An appropriate redundant variant of binary notation also suffices and requires much less space on the storage tape [4].

To simulate more than one counter in real time using a single tape is much harder. For any k, in fact, it is hard to imagine how fewer than k separate tapes can suffice to simulate k counters in real time. Since the contents of the counters to be simulated can fluctuate completely independently, we seem to be forced to consider simulations that actually handle the separate counters separately, say on k separate "tracks" of the one available tape. The problem is to ensure that the simulator's one tape head is always in the right place for every one of these separate handlings, since the next command might be addressed by any of the simulated counters.

Each "separate handling" above is essentially a real-time simulation of one counter. The requirement that the tape head is always in the right place can be formulated most clearly if our counters are "enhanced" to handle one additional command, a command to "do nothing." (Any efficient simulation of an unenhanced counter trivially yields an efficient simulation of an enhanced one, anyway: Simply handle each "do nothing" as if it were an "increment" followed by a "decrement.") Then we can view each command to a multiple-counter storage unit as a tuple of commands, one to each separate counter. What we need, therefore, is a real-time, single-counter simulation that is "oblivious" in the sense that neither its head position nor its times of interaction with the outside world (to respond to commands and to receive new ones) depend at all on the particular command sequence. Our real-time simulation of a k-counter storage unit is indeed based on performing, on a separate track of the one available storage tape, just such a simulation for each of the k simultaneous command streams.

In the rest of this section, without further motivation, we preview the entire oblivious simulation of a single counter. In the following sections, on the other hand, we return to an evolutionary top-down development of the simulation, with each successive refinement motivated by some outstanding inadequacy or loose end. Having previewed the final concrete result, the reader will better appreciate the direction and progress of that evolution.

For transparency, we actually implement our oblivious one-counter simulation on a single-tape Turing machine model that is apparently stronger than the one defined above. The stronger model can write and read symbols from some slightly larger alphabet on its storage tape, and each next step can depend on, change, and shift among all the symbols in some small *neighborhood* of the head position on the storage tape. By coding in binary, and by conceding a somewhat larger (but still fixed) bound on command–response delay time, however, we could straightforwardly replace any such oblivious real-time simulator by an oblivious one of the promised variety.

Each nonblank storage tape symbol used by the simulator includes a base symbol from the set $\{-3, -2, -1, 0, 1, 2, 3, *\}$ and a left or right overarrow. Optionally, it can also include an underline and one or two primes. The purpose of the base symbol $*$ is to mark the position of the read–write head. The initial storage contents is treated as if it were

$$\cdots\ \overleftarrow{0}\ \overleftarrow{0}\ \overleftarrow{0}'\ \overrightarrow{*}\ \overrightarrow{0}\ \overrightarrow{0}\ \overrightarrow{0}\ \cdots.$$

With such a rich storage tape alphabet, our simulator does not have to remember anything in finite-state control—a single state will suffice. Therefore, since even the head position will be implicit in the contents of the storage tape, the transition rules will be just a set of context-sensitive rules for rewriting the storage tape. We promised five such rules, but they are actually five entire *schemes*:

$$\begin{array}{rll}
b\ \overleftarrow{*}\ c' & \Longrightarrow \overrightarrow{*}\ bc & \\
\overrightarrow{b}\ \overleftarrow{*}\ c & \Longrightarrow c'\ \overrightarrow{*}\ \overleftarrow{b}, & \text{propagating into } b\text{, and then from } b \text{ to } c, \\
a\ \overleftarrow{b}\ \overleftarrow{*}\ c & \Longrightarrow ac''\ \overrightarrow{*}\ \overleftarrow{b}, & \text{propagating into } b\text{, and then from } b \text{ to } a, \\
b\ \overleftarrow{*}\ c''\ \overleftarrow{d} & \Longrightarrow b\ \overleftarrow{*}\ \overrightarrow{d}\ c', & \text{propagating from } d \text{ to } c, \\
b\ \overleftarrow{*}\ c''\ \overrightarrow{d}\ e & \Longrightarrow b\ \overleftarrow{*}\ \overrightarrow{d}\ c''e, & \text{propagating from } d \text{ to } e.
\end{array}$$

Each of a, b, c, d, and e can be any member of $\{-3, -2, -1, 0, 1, 2, 3\}$. Except on the symbol with base c, primes are not shown and are unchanged by the transitions. Similarly, arrows not shown are unchanged by the transitions. The mirror-image reflections of the rules describe the transitions when $*$ lies beneath a *right* arrow; thus, for example, the very first transition is according to the first of the five schemes, yielding

$$\cdots\ \overleftarrow{0}\ \overleftarrow{0}\ \overleftarrow{0}\ \overrightarrow{0}\ \overleftarrow{*}\ \overrightarrow{0}\ \overrightarrow{0}\ \cdots.$$

Note that the rule for each next transition will be determined by the number of primes on c and the direction of the arrow over b or d, and that the symbols playing these roles will be determined by the direction of the arrow over $*$. It remains only to give the rules for information "propagation," for maintenance of the underlines (not shown in the rule schemes), and for generation of responses to the commands.

"Propagation from b to c" is essentially a "carry" or "borrow" operation: If b is 3, then reduce it by 4 (to -1) and add 1 to c. If b is -3, then increase it by 4 (to 1) and subtract 1 from c. If either of these actions changes c *to* 0, and c was not underlined, then remove the underline from b; and, if either action changes c *from* 0, and b was not underlined, then add an underline to b. Leave all other underlining unchanged.

"Propagation into b" depends on the next input command. On a command to increment or decrement the counter, b is incremented or decremented accordingly. The result is a count of zero if and only if the resulting b is 0, without an underline.

The delay between the handling of successive input commands is at most three steps, counts of zero are detected correctly, and no base symbol is ever required to overflow past 3 or to underflow past −3, although these facts are not at all clear from just the rules. It *is* clear from the rules that the simulation is both deterministic and oblivious.

As an example, suppose every command is to *increment* the counter. Then the results of the first six transitions are as follows:

$$\begin{array}{ll}
\cdots \overleftarrow{0}\ \overleftarrow{0}\ \overrightarrow{0}\ \ \overleftarrow{*}\ \ \overrightarrow{0}\ \overrightarrow{0}\ \ \overrightarrow{0}\ \overrightarrow{0} \cdots & \text{(by rule 1),} \\
\cdots \overleftarrow{0}\ \overleftarrow{0}\ \overrightarrow{0}'\ \overrightarrow{*}\ \ \overleftarrow{1}\ \overrightarrow{0}\ \ \overrightarrow{0}\ \overrightarrow{0} \cdots & \text{(by rule 2),} \\
\cdots \overleftarrow{0}\ \overleftarrow{0}\ \overrightarrow{0}\ \ \overleftarrow{1}\ \ \overleftarrow{*}\ \overrightarrow{0}\ \ \overrightarrow{0}\ \overrightarrow{0} \cdots & \text{(by rule 1),} \\
\cdots \overleftarrow{0}\ \overleftarrow{0}\ \overrightarrow{0}\ \ \overrightarrow{0}''\ \overrightarrow{*}\ \overleftarrow{2}\ \ \overrightarrow{0}\ \overrightarrow{0} \cdots & \text{(by rule 3),} \\
\cdots \overleftarrow{0}\ \overleftarrow{0}\ \overrightarrow{0}'\ \overleftarrow{0}\ \ \overrightarrow{*}\ \overleftarrow{2}\ \ \overrightarrow{0}\ \overrightarrow{0} \cdots & \text{(by rule 4),} \\
\cdots \overleftarrow{0}\ \overleftarrow{0}\ \overrightarrow{0}'\ \underline{\overrightarrow{-1}}\ \overleftarrow{*}\ \overleftarrow{1}'\ \overrightarrow{0}\ \overrightarrow{0} \cdots & \text{(by rule 2).}
\end{array}$$

Continuing in this way, the result of the first 2,980,000 transitions, including the execution of 1,191,993 commands to increment the counter, is

$$\cdots\ \overleftarrow{0}\ \overleftarrow{0}\ \overrightarrow{0}'\ \overrightarrow{0}'\ \underline{\overrightarrow{0}}\ \overrightarrow{1}''\ \underline{\overrightarrow{0}'}\ \underline{\overrightarrow{2}}\ \underline{\overrightarrow{-1}''}\ \overrightarrow{*}\ \underline{\overleftarrow{1}}\ \underline{\overrightarrow{1}'}\ \underline{\overleftarrow{0}}\ \underline{\overleftarrow{2}''}\ \underline{\overrightarrow{-1}}\ \underline{\overleftarrow{1}}\ \overleftarrow{0}'\ \overleftarrow{0}'\ \overrightarrow{0}\ \overrightarrow{0}\ \cdots.$$

(For now, this should seem pretty obscure as a representation for 1,191,993. It will turn out that the base symbols are a scrambled radix-4 representation for that number:

$$(1021(-1)001(-1)21)_4 = (1191993)_{10}.$$

The unscrambled order is implicit in the arrows and primes. The underlining indicates which radix-4 digits are significant, except that the *leading* significant digit is *not* underlined. (In a radix number, a digit is *significant* as long as it is not a leading 0.))

3. *Oblivious Counting*

There is a relatively familiar technique that makes it possible to maintain a counter obliviously in real time *if* the oblivious order of position access can be nonsequential. The oblivious version [9] of the classical two-tape simulation [8] of multiple Turing machine tapes is based implicitly on the technique. The technique involves a liberalization of ordinary fixed-radix notation, allowing an expanded range of "signed digits" in each position [1, 2]. This, in turn, allows some choice on numbers' representations and some optional delay in carry propagation. To maintain such a representation as the represented number is incremented and decremented, we need only visit the various positions often enough to avoid overflow and underflow. The following two requirements, which are oblivious to the particular sequence of commands, are sufficient for such a scheme to be able to handle commands in real time:

(1) There is a chance ("0-opportunity") to propagate information (increments and decrements) into position 0 at least once every $O(1)$ steps.
(2) There is a chance ("$(i + 1)$-opportunity") to propagate information (carries and borrows) from position i into position $i + 1$ at least once every $O(1)$ times there is an i-opportunity.

These requirements are met, for example, by a schedule that provides a 0-opportunity every other step, a 1-opportunity every other remaining step, a 2-opportunity every other still remaining step, etc.:

0 1 0 2 0 1 0 3 0 1 0 2 0 1 0 4 0 1 0 2 0 1 0 3 0 1 0 2 0 1 0 5 0 1 0 2 0 1 0 3 0 1 0 2 0 1 0 4 0 1 0 2....

This is the sequence of carry propagation distances when we count in binary, so let us call it the *binary carry schedule.*

To see that the requirements suffice, consider using a radix r that is large compared with the constants ("$O(1)$") with which the requirements are satisfied. Symmetrically allow as "digits" all integers d satisfying $-r < d < +r$. (For our ultimate use, the radix $r = 4$ will be large enough; this explains the use of the digit set $\{-3, -2, -1, 0, 1, 2, 3\}$ in Section 2's preview of the simulation.) As suggested in Section 2, maintain an underline beneath each significant digit except for the leading one. Propagate information from position i to position $i + 1$ according to the following simple rules:

"Carry" if the digit is greater than $r/2$.
"Borrow" if the digit is less than $-r/2$.
Do nothing if the digit is bounded by $r/2$ in absolute value.

(To "carry," reduce the digit in position i by r, and increment the digit in position $i + 1$ by 1. To "borrow," reduce the digit in position $i + 1$ by 1, and increment the digit in position i by r.) By induction, the properties of the maintenance schedule ensure that no digit will have to exceed $r - 1$ in absolute value. As a consequence, the only digit that might change from zero to nonzero, or vice versa, is at position $i + 1$ above, so that only the underlining at position i might have to change, and so that correct underlining can be maintained without any additional access to the digits of the counter. As another consequence, the leading significant digit (if there is one) will always correctly indicate the sign of the entire count, so that the count will be 0 only when the frequently observed digit at position 0 is a 0 with no underline.

4. *Permutation for Sequential Access*

With only *sequential* access, it seems impossible to visit the positions of a radix number according to the scheduling requirements above. The first requirement keeps us close to the low-order digit, while the second requirement draws us to arbitrarily high-order digits. This intuition is wrong, however; even with the strictly sequential access available on a single Turing machine tape, we *can* satisfy the requirements. The trick is to maintain, on the main track of the tape, an appropriate dynamically (but obliviously) changing *permutation* of the radix positions. We turn now to a top-down development and implementation of a suitable permutation procedure.

Since the permutation procedure will be oblivious to the actual contents of the radix positions, and since position numbers will greatly clarify the permutation being performed, we speak as if we are permuting the position numbers themselves. It is important to remember, however, that it will be impossible with any finite tape alphabet for our simulator to maintain these unbounded position numbers on its tape without using too much space and time. To recognize what may be obvious from the position numbers, the ultimate simulator will have to maintain appropriate auxiliary markers from some finite alphabet (primes, double primes, and overarrows in the simulation we describe) on an auxiliary track of its tape.

Consider the problem of visiting the positions of a radix number according to the binary carry schedule. The key to the schedule is that it brackets each visit to position $i + 1$ by full "tours" of positions $0, \ldots, i$, denoted by *tour*(i):

```
tour(i + 1):        tour(0):
  tour(i)             visit 0
  visit i + 1
  tour(i)
```

Noting that appending "visit $i + 1$; *tour*(i)" onto the end of *tour*(i) always gives *tour*($i + 1$), we see that *tour*(∞) makes sense:

```
tour(∞):
  visit 0
  visit 1; tour(0)
  visit 2; tour(1)
  visit 3; tour(2)
     ⋮
```

In fact *tour*(∞) is precisely the entire binary carry schedule.

For a Turing machine implementation of all this touring, we must permute to keep the head, represented by $*$ as in Section 2, always near position number 0. Thus we might try the permutational side effect

$$tour(i)\colon * 012 \cdots i(i + 1) \Rightarrow i \cdots 210 * (i + 1)$$

as preparation for the first visit to $i + 1$. But then *tour*(i) (or even its symmetric mirror image) would no longer complete the desired analogous preparation (i.e., *tour*($i + 1$)) for the first visit to $i + 2$. With the latter goal in mind, we are led to push position $i + 1$ left during the second (mirror-image) iteration of *tour*(i) and to introduce into *tour*($i + 1$) a third iteration of *tour*(i), to get back to position $i + 2$. This way, the permutational side effect of *tour*($i + 1$) is from $* 012 \cdots i(i + 1)$ initially, to $i \cdots 210 * (i + 1)$ after the first iteration of *tour*(i), to $(i + 1) * 012 \cdots i$ after the second iteration, finally to $(i + 1)i \cdots 210 *$ after the third iteration, as desired. This leads us to refine our terminology in order to reflect the two variants of *i-tour* (*tour*(i) above):

$$\begin{array}{ll} tour(i, -)\colon * 012 \cdots i \Rightarrow i \cdots 210 * & \\ tour(i, -)\colon i \cdots 210 * \Rightarrow * 012 \cdots i & \text{(mirror image)} \\ tour(i, +)\colon j * 012 \cdots i \Rightarrow i \cdots 210 * j & \\ tour(i, +)\colon i \cdots 210 * j \Rightarrow j * 012 \cdots i & \text{(mirror image)} \end{array}$$

We refer to these variants as *negative* i-tours and *positive* i-tours, respectively, depending on whether some position j *is not* or *is* being "transported." Note that we do not distinguish notationally between a tour and its mirror image, since only one of the two can be applicable at a time, depending on the current location of position 0. Similarly, we do not incorporate into the notation the position j being transported by a positive tour, since there is never any choice.

Suppressing explicit visits now (since convenient i-opportunities will arise at a different point in our scheme, and since the visits do not affect the actual permutation process anyway), we arrive at the following mutually recursive implementations for our evolving tours (the program locations are labeled (a) through (e) for later reference):

```
tour(i + 1, −):      (start with ∗ 012 ⋯ i(i + 1))
        tour(i, −)   (permute to i ⋯ 210 ∗ (i + 1))
    (a) tour(i, +)   (permute to (i + 1) ∗ 012 ⋯ i)
    (b) tour(i, −)   (permute to (i + 1)i ⋯ 210 ∗)

tour(i + 1, +):      (start with j ∗ 012 ⋯ i(i + 1))
        tour(i, +)   (permute to i ⋯ 210 ∗ j(i + 1))
    (c) pushback     (permute to i ⋯ 210 ∗ (i + 1)j)
    (d) tour(i, +)   (permute to (i + 1) ∗ 012 ⋯ ij)
    (e) tour(i, −)   (permute to (i + 1)i ⋯ 210 ∗ j)
```

The recursive strategy for *tour*($i+1$, −) is as described previously, but the strategy for *tour*($i+1$, +) is new. Note that the latter requires a new permutation step, called a *pushback*, to push the nonzero position currently adjacent to the head beyond the next adjacent position. Finally, since appending

(a) *tour*(i, +); (b) *tour*(i, −)

onto the end of *tour*(i, −) always gives *tour*($i+1$, −), we again have a well-defined infinite limit:

```
tour(∞, −):                               (start with ∗ 012345 ⋯)
                            tour(0, −)    (permute to 0 ∗ 12345 ⋯)
    (a) tour(0, +);    (b) tour(0, −)     (permute to 10 ∗ 2345 ⋯)
    (a) tour(1, +);    (b) tour(1, −)     (permute to 210 ∗ 345 ⋯)
    (a) tour(2, +);    (b) tour(2, −)     (permute to 3210 ∗ 45 ⋯)
              ⋮                                    ⋮
```

It is *tour*(∞, −) that we actually implement.

5. *Recursion Elimination*

By induction, the entire permutation process *tour*(∞, −) involves just three, symmetric pairs of atomic moves:

$$\begin{array}{lll} tour(0,-)\colon\ * \, 0 \Longrightarrow 0 \, * & tour(0,+)\colon\ j * 0 \Longrightarrow 0 * j & pushback\colon\ 0 * ji \Longrightarrow 0 * ij \\ \qquad\qquad 0 \, * \Longrightarrow * \, 0 & \qquad\qquad 0 * j \Longrightarrow j * 0 & \qquad\qquad ij * 0 \Longrightarrow ji * 0 \end{array}$$

Our simulator will have to determine which of these local permutations to perform at each step. The problem is analogous to the derivation of a nonrecursive solution to the "Towers of Hanoi" problem from the more obvious recursive solution [3]. In this section we solve the problem by adding a small number of carefully chosen notations to the symbols being permuted.

Because position 0 will always be to the immediate left or right of the head, the simulator will be able to maintain the correct current direction to position 0 in finite-state control, narrowing the possibilities to just one atomic move from each pair above. The remaining problem is to determine whether the next step should be a negative 0-tour, a positive 0-tour, or a pushback.

OBSERVATION 1. *For every i, the respective first moves of tour(i, −) and tour(i, +) are tour(0, −) and tour(0, +).*

Except for the initial situation, when *tour*(0, −) is required explicitly, program locations (a)–(e) account for all situations. By Observation 1, it will suffice always to know whether the next move starts a negative tour (program locations (b), (e)), starts a positive tour (program locations (a), (d)), or is a pushback (program location (c)). A good clue would be the largest action that the *previous* move *ended*; this clue is not readily available, however, since negative $(i + 1)$-tours and positive $(i + 1)$-tours both end with the same move, a negative 0-tour.

OBSERVATION 2. *A positive tour ends with the head adjacent to the transported position j.*

OBSERVATION 3. *By induction, at no time properly within a tour is the head adjacent to a position not explicitly involved in the tour. (The positions explicitly involved in tour(i, −) are* 0 *through i, and the ones explicitly involved in tour(i, +) are these and also the transported position j.)*

COROLLARY. *The position j that gets pushed back at the outermost level of a positive tour will next be adjacent to the head at the end of that positive tour.*

This last corollary presents an opportunity to recognize the end of a positive tour: The head can leave a "message" attached to the position that gets pushed back, indicating that a positive tour is in progress. (In our ultimate implementation, the messages will be single and double primes on symbols.) Consequently, the simulator will be able to recognize when a positive tour is ending, at which time it can delete the message (remove the single or double prime). (In the special case of the one-move positive *tour*(0, +), there is no pushback; in this case, for uniformity, the same sort of message can be attached to the relevant position *j*, in the one move that does take place.) The absence of such a message, therefore, will surely indicate program location (a) or (d) and hence that the next move should be *tour*(0, +). In the *presence* of such a message, however, it still remains to distinguish program location (c) (which is followed by a pushback) from program locations (b) and (e) (which are followed by *tour*(0, −)). For this purpose, we introduce an auxiliary distinction between two varieties of positive tour, a distinction that we try to record as part of the message corresponding to the positive tour. The distinction is simply $j = i + 1$ versus $j > i + 1$:

$$\begin{aligned}
&tour'(i, +)\text{: } (i + 1) * 012 \cdots i \Rightarrow i \cdots 210 * (i + 1)\\
&tour'(i, +)\text{: } i \cdots 210 * (i + 1) \Rightarrow (i + 1) * 012 \cdots i\\
&tour''(i, +)\text{: } \quad j * 012 \cdots i \Rightarrow i \cdots 210 * j \qquad (j > i + 1)\\
&tour''(i, +)\text{: } \quad i \cdots 210 * j \Rightarrow j * 012 \cdots i \qquad (j > i + 1)
\end{aligned}$$

In the correspondingly revised recursion, doubly primed positive tours are needed only for the first subtour at the outermost level of each positive tour. Because different messages have to be left, we begin now to distinguish between singly and doubly primed pushbacks. For use in our analysis, we add the recursion level of a pushback to the notation, even though it is not algorithmically significant.

```
tour(i + 1, −);                  (start with * 012 ··· i(i + 1))
        tour(i, −)               (permute to i ··· 210 * (i + 1))
    (a) tour′(i, +)              (permute to (i + 1) * 012 ··· i)
    (b) tour(i, −)               (permute to (i + 1)i ··· 210 *),

tour′(i + 1, +):                 (start with (i + 2) * 012 ··· i(i + 1))
        tour″(i, +)              (permute to i ··· 210 * (i + 2)(i + 1))
    (c) pushback′(i + 1)         (permute to i ··· 210 * (i + 1)(i + 2))
    (d) tour′(i, +)              (permute to (i + 1) * 012 ··· i(i + 2))
    (e) tour(i, −)               (permute to (i + 1)i ··· 210 * (i + 2)),

tour″(i + 1, +):                 (start with j * 012 ··· i(i + 1))
        tour″(i, +)              (permute to i ··· 210 * j(i + 1))
    (c) pushback″(i + 1)         (permute to i ··· 210 * (i + 1)j)
    (d) tour′(i, +)              (permute to (i + 1) * 012 ··· ij)
    (e) tour(i, −)               (permute to (i + 1)i ··· 210 * j),

tour(∞, −):                      (start with * 012345 ···)
                      tour(0, −) (permute to 0 * 12345 ··· )
    (a) tour′(0, +); (b) tour(0, −) (permute to 10 * 2345 ···)
    (a) tour′(1, +); (b) tour(1, −) (permute to 210 * 345 ···)
    (a) tour′(2, +); (b) tour(2, −) (permute to 3210 * 45 ···)
                         ⋮                      ⋮
```

As desired, now, the end of the doubly primed variety of positive tour will indicate program location (c), and the end of the singly primed variety will indicate program location (b) or (e).

It remains to find a way to recognize of which variety of positive tour each pushback is a top-level part, and which is the variety of each positive 0-tour, so that the right messages (single or double prime, corresponding to the singly or doubly primed variety of pushback or positive tour) can be recorded. For these purposes, we maintain with each position the direction in the current permutation to its successor. (This is the purpose of the overarrows.) When we summarize in Section 7, we indicate how to keep this information up-to-date. To see that this directional information will help, we need one more inductive observation:

OBSERVATION 4. *In each invocation of tour″$(i, +)$ (only two possibilities above), the first uninvolved position initially beyond position i is position $i + 1$. (In either case, the initial permutation will include $j * 012 \cdots i(i + 1)$ or its mirror image.)*

In all our invocations of *tour″*$(i + 1, +)$, therefore, the first uninvolved position initially beyond position $i + 1$ will be position $i + 2$, so that the precondition for *pushback″*$(i + 1)$ will always be $0 * j(i + 1)(i + 2)$ (or its mirror image). The precondition for *pushback′*$(i + 1)$, on the other hand, will always be $0 * (i + 2)(i + 1)$ (or its mirror image). The distinction can be recognized from the directional information for position $i + 1$. Similarly, the precondition for *tour″*$(0, +)$ will always be $10 * j$ (or its mirror image), while the precondition for *tour′*$(0, +)$ will always be $0 * 1$ (or its mirror image), a distinction that can be recognized from the directional information for position 0.

In summary, here are suitable specifications for the evolved versions of all the tours and pushbacks (except for mirror images), now showing single- and double-prime messages (but not showing overarrows, since we are still showing explicit position numbers):

$$\begin{array}{rll}
\mathit{tour}(i, -)\colon & {}' * 012 \cdots i \Rightarrow i \cdots 210 * & \\
\mathit{tour}'(i, +)\colon & (i + 1) * 012 \cdots i \Rightarrow i \cdots 210 * (i + 1)' & \\
\mathit{tour}''(i, +)\colon & j * 012 \cdots i(i + 1) \Rightarrow i \cdots 210 * j''(i + 1) & (j > i + 1) \\
\mathit{pushback}'(i)\colon & 0 * (i + 1)''i \Rightarrow 0 * i(i + 1)' & \\
\mathit{pushback}''(i)\colon & 0 * j''i(i + 1) \Rightarrow 0 * ij''(i + 1) & (j > i + 1)
\end{array}$$

It is easy to check inductively that the recursive implementations do maintain the specifications:

$\mathit{tour}(i + 1, -)$:	(start with $' * 012 \cdots i(i + 1)$)
$\quad\mathit{tour}(i, -)$	(permute to $i \cdots 210 * (i + 1)$)
$\quad\mathit{tour}'(i, +)$	(permute to $(i + 1)' * 012 \cdots i$)
$\quad\mathit{tour}(i, -)$	(permute to $(i + 1)i \cdots 210 *$)
$\mathit{tour}'(i + 1, +)$:	(start with $(i + 2) * 012 \cdots i(i + 1)$)
$\quad\mathit{tour}''(i, +)$	(permute to $i \cdots 210 * (i + 2)''(i + 1)$)
$\quad\mathit{pushback}'(i + 1)$	(permute to $i \cdots 210 * (i + 1)(i + 2)'$)
$\quad\mathit{tour}'(i, +)$	(permute to $(i + 1)' * 012 \cdots i(i + 2)'$)
$\quad\mathit{tour}(i, -)$	(permute to $(i + 1)i \cdots 210 * (i + 2)'$)
$\mathit{tour}''(i + 1, +)$:	(start with $j * 012 \cdots i(i + 1)(i + 2)$)
$\quad\mathit{tour}''(i, +)$	(permute to $i \cdots 210 * j''(i + 1)(i + 2)$)
$\quad\mathit{pushback}''(i + 1)$	(permute to $i \cdots 210 * (i + 1)j''(i + 2)$)
$\quad\mathit{tour}'(i, +)$	(permute to $(i + 1)' * 012 \cdots ij''(i + 2)$)
$\quad\mathit{tour}(i, -)$	(permute to $(i + 1)i \cdots 210 * j''(i + 2)$)
$\mathit{tour}(\infty, -)$:	(start with $' * 012345 \cdots$)
$\quad\mathit{tour}(0, -)$	(permute to $0 * 12345 \cdots$)
$\quad\mathit{tour}'(0, +)$; $\mathit{tour}(0, -)$	(permute to $10 * 2345 \cdots$)
$\quad\mathit{tour}'(1, +)$; $\mathit{tour}(1, -)$	(permute to $210 * 345 \cdots$)
$\quad\mathit{tour}'(2, +)$; $\mathit{tour}(2, -)$	(permute to $3210 * 45 \cdots$)
$\vdots$	$\vdots$

6. *Opportunities to Carry and to Borrow*

We see from the above preconditions for $\mathit{pushback}'(i + 1)$ and $\mathit{pushback}''(i + 1)$ (the "$(i + 1)$-pushbacks") that these operations can serve as $(i + 2)$-opportunities. Similarly, $\mathit{tour}'(0, +)$ and $\mathit{tour}''(0, +)$ (the "positive 0-tours") can serve as 1-opportunities. Since the head is always adjacent to position 0, every step is a good time to propagate increments and decrements into position 0; if we designate only the positive 0-tours as 0-opportunities, however, we will ultimately be able to choose a slightly smaller radix for our notation.

OBSERVATION 5. *If we omit j-pushbacks for $j > i$, then $\mathit{tour}(\infty, -)$ is an infinite concatenation of negative and positive i-tours, the first of which is negative, the second of which is positive, and no three consecutive of which are all negative* *or all positive. (To see the last part, make the analogous observation by induction on $i' \geq i$ for each negative and positive i'-tour, and finally note that tour$(\infty, -)$ is the limit of the negative tours.)*

COROLLARY. *In tour$(\infty, -)$, our information propagation requirements are satisfied with respective constants* 3 *and* 4:

(1) *There is a* 0*-opportunity at least once every three steps.*
(2) *There is a* 1*-opportunity every time there is a* 0*-opportunity.*
(3) *There are exactly two $(i + 1)$-opportunities before the first $(i + 2)$-opportunity, and at most four $(i + 1)$-opportunities between $(i + 2)$-opportunities.*

PROOF OF THIRD PART. The $(i + 1)$-opportunities are distributed one in each positive i-tour. Using Observation 5 to focus on $(i + 1)$-tours, therefore, we see that each negative tour presents one $(i + 1)$-opportunity and no $(i + 2)$-opportunity, and that each positive tour presents one $(i + 1)$-opportunity before its one $(i + 2)$-opportunity, and one after. The two initial $(i + 1)$-opportunities come from the initial negative and positive tours, and the maximum of four intervening $(i + 1)$-opportunities arise when a consecutive pair of negative tours is bracketed by positive tours. □

Since $5 + 4 \leq 10 - 1$, it follows that $r = 10$ will be a large enough radix. The more careful analysis in Section 8 reveals that even $r = 4$ will be large enough.

7. *Formal Summary*

In Section 5 we showed how to annotate the symbols being permuted in the recursively defined *tour*$(\infty, -)$ in such a way that the very same permutation can be carried out nonrecursively by a deterministic single-tape Turing machine, based entirely on local cues. In Section 6 we observed that the same annotations provide sufficient cues for adequate opportunities to perform the increments, decrements, carries, and borrows required for our real-time simulation of a counter. In this section we finally relate all this to the few simple rules previewed in Section 2.

For transparency, we summarize the rules we have derived in three increasingly formal stages. In increasing order of difficulty, the four main cases are the first move, the case in which a single-prime message is received, the case in which no message is received, and the case in which a double-prime message is received. The first move is always *tour*$(0, -)$. When a single-prime message is received, *tour*$(0, -)$ is again the correct move. When no message is received, the correct move is either *tour*$'(0, +)$ or *tour*$''(0, +)$, depending on whether position 1 is adjacent to the head or beyond position 0; either way, a carry or borrow can be propagated as described above, and an indicative message should be left with the transported position. When a double-prime message is received, the correct move is a singly or doubly primed pushback, depending on directional information near the head as described above; either way, a carry or borrow can be propagated as described above, and an indicative message should be left with the position that is pushed back.

In the second stage, we reformulate our summary via formal rules in terms of position numbers. For the messages corresponding to completion of singly and doubly primed positive tours, we use single and double primes on the position numbers. Except in the case of the special rule for the very first move ($* \; 0 \Rightarrow 0 \; *$), the mirror image of each rule is also a rule; so we list only rules with position 0 initially to the *left* of the head. Only on the *other* side of the head do we show primes explicitly, since these primes constitute the message being received.

(1) *Single-prime message:* negative 0-tour

$$0 * i' \Rightarrow * 0\ i.$$

(2) *no message:* positive 0-tour

$$\begin{aligned} 0 * 1 &\Rightarrow 1' * 0, && \text{propagate into 0 and then from 0 to 1,} \\ 1\ 0 * j &\Rightarrow 1\ j'' * 0, && \text{propagate into 0 and then from 0 to 1 } (j > 1). \end{aligned}$$

(3) *Double-prime message:* pushback

$$\begin{aligned} 0 * (i+2)''(i+1) &\Rightarrow 0 * (i+1)(i+2)', && \text{propagate from } i+1 \text{ to } i+2, \\ 0 * j''(i+1)(i+2) &\Rightarrow 0 * (i+1)j''(i+2), && \text{propagate from } i+1 \text{ to } i+2\ (j > i+2). \end{aligned}$$

In our final, unavoidably obscure reformulation, we replace the position numbers with nonnumeric base-symbol variables and the overarrows that are actually present. For base-symbol variables whose overarrows are irrelevant and do not change, however, we omit the explicit overarrows from the rules. To avoid explicit reference to finite-state control, we replace the head marker $*$ with either $\overleftarrow{*}$ or $\overrightarrow{*}$ to indicate whether position 0 is just to the left or just to the right. Except for the start rule ($\overrightarrow{*}\ a \Rightarrow a\ \overleftarrow{*}$), each rule again has an implicit symmetric rule.

$$\begin{aligned} b\ \overleftarrow{*}\ c' &\Rightarrow \overrightarrow{*}\ bc \\[1ex] \overrightarrow{b}\ \overleftarrow{*}\ c &\Rightarrow c'\ \overrightarrow{*}\ \overleftarrow{b}, && \text{propagating into } b \text{, and then from } b \text{ to } c, \\ a\ \overleftarrow{b}\ \overleftarrow{*}\ c &\Rightarrow ac''\ \overrightarrow{*}\ \overleftarrow{b}, && \text{propagating into } b \text{, and then from } b \text{ to } a, \\[1ex] b\ \overleftarrow{*}\ c''\ \overleftarrow{d} &\Rightarrow b\ \overleftarrow{*}\ \overrightarrow{d}\ c', && \text{propagating from } d \text{ to } c, \\ b\ \overleftarrow{*}\ c''\ \overrightarrow{d}\ e &\Rightarrow b\ \overleftarrow{*}\ \overrightarrow{d}\ c''\ e, && \text{propagating from } d \text{ to } e. \end{aligned}$$

The start rule closely resembles (the mirror image of) the first of the five more general rules. If we initially provide a singly primed "endmarker" to the left of the head, then the separate start rule actually does become redundant; the result, at least if we use radix $r = 4$, is the simulation previewed in Section 2.

8. *Space Analysis*

The space used for the first n steps of the *most* space-efficient simulation of k counters is within an additive constant of $k \log_2 n$ bits, in the worst case. For k large, we see now that a straightforward implementation of our real-time, oblivious simulation requires only about 2.5 times this much space.

Regardless of the particular (large enough) radix used, the number of distinct *positions* involved by step n in the permutation process is within an additive constant of $\log_3 n$. To see this, note that the process first reaches position $i + 1$ at the end of *tour*(i, $-$), and that the number of steps in a negative i-tour is exactly $\frac{5}{4}3^i - \frac{1}{2}i - \frac{1}{4}$. The latter, along with the fact that the number of steps in a *positive* i-tour is exactly $\frac{5}{4}3^i + \frac{1}{2}i - \frac{1}{4}$, can be proved by straightforward simultaneous induction.

To minimize the space used for each position, we should choose the smallest radix that works. The analysis below shows that 4 works. For each additional counter, therefore, the space needed for each involved position is at most $4 = \lceil \log_2(7 \cdot 2) \rceil$ bits. (Each of the seven signed digits has two versions, one underlined and one not underlined.) The additional, counter-independent space needed for each position is at most $3 = \lceil \log_2(3 \cdot 2) \rceil$ bits. (The message can be absent, a single prime, or a double prime; and the overarrow can point to the left or to the right.) All together, therefore, the space used through step n can be bounded by $(3 + 4k)\log_3 n \approx (1.89 + 2.52k)\log_2 n$ bits.

It remains only to show that no overflow (past 3) or underflow (past -3) will occur if we use 4 as the radix in our simulation. Until an overflow or underflow does occur, each $(i + 1)$-opportunity (and also the implicit initialization) will leave each signed digit in position i in the range from -2 to 2. Therefore, it suffices to show that, although there might be as many as four i-opportunities without an intervening $(i + 1)$-opportunity, at most one of these can actually result in a carry (or, symmetrically, in a borrow).

LEMMA 1. *For each $i \geq 1$, at most one i-opportunity in four can result in a carry. For each $i \geq 0$, therefore, an $(i + 1)$-opportunity intervenes between every pair of increments to the signed digit in position i. (Similarly for borrows and decrements, by symmetry.)*

PROOF. For each $i \geq 1$, the second assertion follows from the first by the third part of the corollary to Observation 5. For $i = 0$, the second assertion is an immediate consequence of the second part of the same corollary.

The proof of the first assertion is by induction on $i \geq 1$, and the general induction step is itself an induction on time. Consider the first or next i-opportunity that results in a carry. This carry leaves the signed digit $-1 = 3 - 4$ in position $i - 1$. By the (second) assertion for $i - 1$, this can increase to at most 0 by the next i-opportunity, to at most 1 by the third i-opportunity, and to at most 2 by the fourth i-opportunity, none of which requires a carry. □

9. *Further Optimization*

Our overriding objective so far has been to keep the simulation simple. At the expense of some clarity, however, we can make the simulation even more efficient.

There is one easy way to save space in the simulation as presented above. Positions of the separate representations and positions that are adjacent in the current permutation need not be encoded separately. By suitable encoding, therefore, the space used can be kept arbitrarily close to the unrounded limit $(\log_2 6 + k \log_2 14)\log_3 n \approx (1.63 + 2.40k)\log_2 n$.

A more subtle observation leads to saving even more space. Because each radix-4 signed digit is bounded by 3 in absolute value, the number of *significant* signed digits in each counter's representation stays within an additive constant of the *base*-4 logarithm of the counter's contents. With care, therefore, we might hope to limit the number of positions involved in our simulation to the base-4 logarithm of the largest counter contents so far. Even in the worst case that the largest counter contents after the first n steps is n, this would reduce space usage by a factor of $\log_4 n/\log_3 n \approx 0.79$.

One way to take advantage of this potential is to insert some extra pairs of negative i-tours right before the positive i-tour that first transports and involves position $i + 1$. (The first half of each such pair permutes from $i \cdots 210 * (i + 1)(i + 2) \cdots$ back to the original configuration $* 012 \cdots i(i + 1)(i + 2) \cdots$, and the second half permutes up to $i \cdots 210 * (i + 1)(i + 2) \cdots$ again.) To do this, we need only decide at the time we would normally first involve a new position $i + 1$ (with a positive 0-tour) whether to start a negative i-tour instead (with a *negative* 0-tour). We want to involve position $i + 1$ if and only if a significant signed digit is already within a few positions of position $i + 1$.

For this, we need a second version of each uninvolved position, to indicate whether the position is "ripe" for involvement, and we need appropriate opportunities to mark uninvolved positions ripe. If $i + 1$ is the first uninvolved position, then such an opportunity arises each time we reach the configuration $(i - 3) \cdots 210 * (i - 2)(i - 1)i(i + 1) \cdots$, say. It follows from the easy-to-check inductive observation below that directional information will suffice to identify this situation unambiguously. If significance has already reached position $i - 2$, say, then position $i + 1$ can be marked finally as ripe for involvement, and it will become involved in time to receive the first carry from position i.

OBSERVATION 6. *At any time in the permutation process, if $\vec{a}b$ occurs anywhere to the right of the head (or, symmetrically, if $b\overleftarrow{a}$ occurs anywhere to the left of the head), with a prime or double-prime message attached to neither a nor b, and if the position number of a is i, then the position number of b is $i + 1$.*

At the expense of obliviousness, this yields a real-time multicounter simulation that uses space only logarithmic in the *maximum* counter contents. It reduces the worst-case space for a real-time simulation of k counters to about $(\log_2 6 + k \log_2 15)\log_4 n \approx (1.29 + 1.95k)\log_2 n$.

Although, with a slightly different designation of i-opportunities, we could reduce the radix for our simulation's radix notation down to 3, it turns out to be more space efficient to use a *larger* radix. At the mere expense of additional control states, this will reduce the number of bits used for underlines, messages, and overarrows. Repeating the analysis sketched above, but now for an arbitrary radix r, we get a space bound of

$$(\log_2 6 + k \log_2(2(2r - 1) + 1))\log_r n \leq \left(k + \frac{\log_2 6 + 2k}{\log_2 r}\right) \log_2 n.$$

For each $\epsilon > 0$, therefore, we can use a radix r so large that $(k + \epsilon)\log_2 n$ bits will suffice for every k.

Note that the analyses above do give improved results even for oblivious simulation. Since the counter with the largest contents determines head motion, the simulator will be oblivious if it simulates one extra dominant counter of its own, incrementing it at every step. This yields a space bound of $(k + 1 + \epsilon)\log_2 n$ bits for oblivious real-time simulation of k counters.

ACKNOWLEDGMENTS. We thank Laura Sanchis for criticizing an early draft of the paper and Janie Irwin for providing references [1] and [2]. Péter Gács first suggested that the space used by a nonoblivious version of the simulation could be kept proportional to the logarithm of the largest counter contents. The result of the first 2,980,000 transitions was obtained by an implementation of the simulation in PC Scheme, and later checked independently by John Tromp's implementation in C.

REFERENCES

1. ATKINS, D. E. Introduction to the role of redundancy in computer arithmetic. *Computer 8*, 6 (June 1975), 74–77.
2. AVIZIENIS, A. Signed-digit number representations for fast parallel arithmetic. *IRE Trans. Electron. Comput. EC-10*, 3 (Sept. 1961), 389–400.
3. CULL, P., AND ECKLUND, JR., E. F. Towers of Hanoi and analysis of algorithms. *Am. Math. Mon. 92*, 6 (June–July 1985), 407–420.

4. FISCHER, M. J., AND ROSENBERG, A. L. Real-time solutions of the origin-crossing problem. *Math. Syst. Theory 2*, 3 (Sept. 1968), 257–263.
5. FISCHER, P. C., MEYER, A. R., AND ROSENBERG, A. L. Counter machines and counter languages. *Math. Syst. Theory 2*, 3 (Sept. 1968), 265–283.
6. FISCHER, P. C., MEYER, A. R., AND ROSENBERG, A. L. Time-restricted sequence generation. *J. Comput. Syst. Sci. 4*, 1 (Feb. 1970), 50–73.
7. FÜRER, M. Data structures for distributed counting. *J. Comput. Syst. Sci. 28*, 2 (Apr. 1984), 231–243.
8. HENNIE, F. C., AND STEARNS, R. E. Two-tape simulation of multitape Turing machines. *J. ACM 13*, 4 (Oct. 1966), 533–546.
9. PIPPENGER, N., AND FISCHER, M. J. Relations among complexity measures. *J. ACM 26*, 2 (Apr. 1979), 361–381.
10. VITÁNYI, P. M. B. An optimal simulation of counter machines. *SIAM J. Comput. 14*, 1 (Feb. 1985), 1–33.
11. VITÁNYI, P. M. B. An optimal simulation of counter machines: The ACM case. *SIAM J. Comput. 14*, 1 (Feb. 1985), 34–40.

RECEIVED AUGUST 1985; REVISED SEPTEMBER 1987; ACCEPTED MARCH 1988