INDIRECT ADDRESSING AND THE TIME RELATIONSHIPS OF SOME MODELS OF SEQUENTIAL COMPUTATION

PATRICK W. DYMond
Department of Computer Science, University of Toronto, Toronto, Canada

Communicated by Allan Borodin
(Received January 1979)

Abstract—We study the time relationships between several models of computation (variants of counter machines, Turing machines, and random access machines). It is shown that counter machines augmented by a "copy" instruction can be simulated in linear time by counter machines without such an instruction, and that these counter machines can be simulated by RAM's with speedup by a fixed polynomial. Since the difference between augmented counter machines and RAM's lies partly in the latter's indirect addressing capabilities, we obtain bounds on the extent to which these capabilities speed up computations. We also show that unit-cost RAM's can simulate multi-dimensional Turing machines with speedup using their addressing capabilities to efficiently implement multidimensional arrays. Evidence is presented to show that on a restricted class of RAM's, "successor" RAM's, efficient implementation of multi-dimensional arrays is not possible.

1. INTRODUCTION

Turing[1] introduced the idea of a mathematical model of computation, a precise description of an abstract computing machine. Many other models, all capable of performing precisely the same tasks, have since been proposed, and we find them interesting because they sometimes seem to require different amounts of some resource (such as time or storage) to accomplish the same objective. If the difficulty of a task varies from model to model we can ask what features of the models make the task easier or more difficult; and we can also ask about the complexity of such a task on a real computer.

The fact that many different models all are capable of computing exactly the same class of functions is taken as evidence in support of Church's thesis[2]—that this class of functions consists exactly of those functions which are "effectively computable". But from the viewpoint of computational complexity, different models may vary in resources required to accomplish identical tasks; and we have not yet found a general analogue to Church's thesis in this area. Such an analogue might be a claim that the class of functions computed by a particular model $\mathcal{M}$ in any time bound $T$ consists precisely of the functions which are "really computable in sequential (resp. parallel) time $T'$", and that therefore $\mathcal{M}$ is the right model with which to study sequential (resp. parallel) time complexity. A more limited form of this analogue to Church's thesis is the widely-held belief that the Turing machine (described in Section 2) computes in time bounded by a polynomial in the length of the input precisely those functions "really sequentially computable in polynomial time". The fact that many other sequential models compute the same class of functions as the Turing machine in polynomial time is taken as evidence for this belief.

We would like to further develop complexity analogues to Church's thesis, analogues such as:

For any $c \geq 0$, the class of functions computed by a unit-cost successor RAM (described in Section 3) in time $O(n^c)$ where $n$ is the length of the input, is precisely that class of functions "really computable in sequential time $O(n^c)$"—and thus the unit-cost successor RAM is a good model for studying sequential polynomial time complexity.

Evidence for this statement may perhaps be found by studying the relationship between this and other models and by understanding more fully the model's exact power. (On the other hand, the intuitive evidence for Church's thesis is very persuasive because widely differing and extremely general approaches to the problem have all produced the same results; at this time we do not know what form that equally persuasive evidence for analogues dealing with specific
polynomial (sequential) time bounds should take. In fact, it could be conjectured that sequential
time may be too unstable a notion for such specific analogues to exist.)

In this paper we discuss the relative time complexities of several sequential models—Turing
machines, counter machines and random access machines—and examine the power of indirect
addressing in the RAM model.

A *machine model* is a class of formal machines all of whose members have the same
“permissible operations”, “storage predicates”, input-output conventions and storage structure.
“Permissible operations” describe the ways in which storage is modified; “storage predicates”
extract information from the contents of storage. The input-output conventions for the models
discussed in this paper are as follows:

- The input is a word $w$ of unbounded but finite length $n$ composed of symbols from a finite
  alphabet $I$, and is viewed as being written from left to right on $n$ successive squares of an
  *input tape*, with a special symbol $\epsilon \in I$ as a delimiter at each end. Initially the read-only
  input tape head is located at the first symbol of the input, and at each time instance during
  the course of the computation may be moved one square right or left or left unmoved. (If
  the model has only one-way input, the head may only move right.)

- The output is written onto successive squares of the *output tape*, which is infinitely long
to the right; at any time during the course of a computation the machine may write an
  output symbol (some member of the finite output alphabet $O$) on the current tape square
  and move the output tape head one square to the right.

Machine models are described in terms of either *programs* or *transition functions*. When a
transition function is used the model has a *finite state control*; and by a *configuration* of such a
machine we mean the contents of its storage, the state and the input tape contents and head
position. A reduced configuration consists of the state and symbol scanned by the input tape
head and the results of applying the storage predicates to the contents of storage. (The
predicates describe the aspects of the current configuration which affect the transition func-
tion—for Turing machines they identify the symbols currently scanned by the work tape heads,
for counter machines they identify the counters containing zeroes.) The transition function $\delta$
maps a reduced configuration to a (possibly) new state, a subset of the model-specific
permissible operations which may modify the contents of storage, and a set of I/O instructions.

By a *step* of such a machine we mean an application of $\delta$ to the reduced configuration
corresponding to the current total configuration. A *computation* is a sequence of configurations
starting from the *initial configuration*, which may terminate when a halting state is reached or
when the current reduced configuration is not in the domain of the transition function.

When, instead of a transition function, a *program* is used in the description of a model, no
finite state control is needed; instead explicit branching instructions to locations in the program
serve this function. A program consists of a sequence of (possibly labelled) statements, each of
which can be:

- The instruction HALT.
- One of the model-specific permissible operations on storage.
- An input or output instruction.
- The word IF, followed by a storage predicate, followed by a statement.
- An instruction to branch to a specific instruction.

A computation then consists of a sequence of statement executions.

The measure of the time used by a machine depends on the model being considered—for
some models it can just be the number of steps in the computation—but since other measures
are sometimes useful, we associate with each model timing instructions from which the number
of time units required for a given step may be calculated.

We can view machines in several ways—as *acceptors* of subsets of $I^*$, as *generators* of
sequences, or as computers of partial functions from $I^*$ to $O^*$. We describe the notion of time
used in this paper in terms of the computation of functions; similar definitions can be framed
for the other viewpoints.

Let $T$ be a function $T: N \to N$ and let $M$ be a machine computing a partial function
$f: I^* \to O^*$. If for all words $w \in \text{dom } f$, $M$ computes $f(w)$ using less than or equal to $T(n)$ time
units, where $n = \max(|w|, |f(w)|)$, we say $M$ works in time $T$, or $M$ computes $f$ in time $T$. If
there exists a constant $c$, such that for all $n$, $T(n) < c \cdot n$, we say $M$ computes $f$ in linear time. If
there exists a constant $c$ such that $M$ takes at most time $c$ between each movement of its (one-way) input head, printing an output symbol between each movement of the input head, and halting within time $c$ of the last movement of the input head, we say $M$ works in real time.

(Note: This definition differs from that of some writers in which real time is taken to mean $T(n) = n$. For Turing machines the two definitions are equivalent because of the well-known "speedup" theorem of [3]. For other models, where corresponding "speedup" theorems are not known, the more general definition of real time used in this paper may not be equivalent to the restricted definition; our definition is the more widely accepted one, particularly when referring to simulations.)

Models can be studied both by examining their individual properties and by comparing them to other models. For these comparisons we use the concept of simulation. We say that machine $M$ simulates machine $M'$ if they both compute the same partial function. We say a simulation is constructive if the computation of the simulating machine $M$ can be viewed as a sequence of stages, corresponding to the steps of $M'$, the machine being simulated. Additionally, in a constructive simulation at any stage we can effectively determine the configuration of $M'$ given that of $M$.

All of the simulations presented in this paper are constructive in this sense.

Let $T_1$, $T_2$ be functions $T_1: N \rightarrow N$, $T_2: N \rightarrow N$. $M$ simulates $M'$ in time $T_1(n)$ when, if $M'$ computes the partial function $f$ in time $T_2(n)$, $M$ computes $f$ in time $T_1(T_2(n))$. The simulation is called linear if there exists a $c \in N$ such that $T_1(n) \leq c \cdot n$ for all $n$. $M$ simulates $M'$ in real time if $M$ takes time $\leq c$ simulating each step of $M'$. We say a model $M_1$ simulates a model $M_2$ in time $T$ if, for every machine $M_2 \in M_2$, we can effectively find a machine $M_1 \in M_1$ which simulates $M_2$ in time $T$.

In the final analysis, additional criteria will be used to evaluate a particular model besides its time relationships with other models. It is desirable that our models be chosen so that their operations and predicates correspond to our notions of what can be done by a real computing agent (human or machine), and that the defined complexities of the operations relate consistently to the complexity of corresponding operations in the real world. But because our models possess an unbounded amount of storage, it is not readily apparent what the precise relationships between their architectures and those of real computers should be.

In this paper we restrict our attention to time relationships of several models. In Section 2, we discuss Turing machines and counter machines. Section 3 examines the random access machine model.

### 2. TURING MACHINES AND COUNTER MACHINES

The Turing machine, first described in Turing[1], has been extensively studied. Many of the relevant theorems as well as programming examples and formal definitions for various Tm models may be found in Hopcroft and Ullman[4]. Informally, a $k$-Tm ($k$-tape Turing machine) consists of a finite state control, input and output tapes and $k$ storage tapes, each infinitely long in both directions, divided into (initially blank) squares each of which may hold a single symbol from the finite worktape alphabet. At any time each of the $k$ storage tape heads sees the symbol written on the current square and in one step may write a new symbol and be moved at most one square left or right.

Because of the limited sequential storage access methods of the Turing machine, some theorems exist for them which have no analogues for more powerful models (such as the RAM). For example, Hopcroft, Valiant and Paul[5] have shown that for any function $T(n) \geq n$, any set which can be accepted by a Tm in time $T \log T$ can be accepted using space only $O(T)$, and thus that space is a more powerful resource than time for Tm's. But many aspects of the model remain to be studied; it is not even known if, for $k \geq 2$, a $k + 1$-tape Tm working in time $T$ is more powerful than a $k$-tape Tm working in time $T$, except for real-time computations. Even for $k = 1$, nothing is known for $T(n) \geq n^2$.

Various "improvements" to the Turing machine have been suggested, among them:

- The fast-rewind Tm, which, on each tape, has a fixed reset square to which the head may jump in a single step.
- The multi-head Tm, on which multiple tape heads co-exist on each work tape.
- The jump Tm, a multi-head Tm on which any head may jump across the tape to the position of any other head in one step.
The two-dimensional Tm, which has storage planes instead of storage tapes; this model can be further generalized to $k$-dimensional storage spaces.

Fischer and Rosenberg [6] have shown that any fast-rewind Tm can be simulated in real time by an ordinary multitape Tm (with more tapes); Fischer et al. [7] have shown the same for the multi-head Tm. Recently Savitch and Vitanyi [8] have shown that the jump-Tm can be simulated in linear time by an ordinary Tm (see Theorem 2.2). A multidimensional Tm of dimension $d$ working in time $T$ can be simulated by an ordinary Tm in time $O(T^{2-1/d})$ [9].

The counter machine [10] is a simply described model which has been studied extensively by Fischer et al. [11]. For real-time language recognition problems, Fischer and Rosenberg [12] have shown one-tape Turing machines to be more powerful than any counter machine.

Informally, a $k$ counter machine ($k$-CM) can be viewed as having a read-only input tape, a write-only output tape, a finite state control and a set of $k$ counters. The only operations permitted on the counters are addition and subtraction of one; the only tests which can be made are tests to see which counters contain zero. A precise description of a $k$-CM consists of:

- A finite set of states, $Q$.
- A finite input alphabet, $I$.
- A set $A \subseteq Q$ of accepting states.
- An initial state $q \in Q$.
- A finite output alphabet $O$.
- A transition function

$$\delta: Q \times I \times \{0, 1\}^k \rightarrow Q \times \{+1, -1, 0\}^k \times (O \cup \{\phi\}) \times H$$

where $H$ is a set of input head movement instructions. $H = \{R, L, N\}$ for a 2-way CM; $H = \{R, N\}$ for a one-way CM.

Initially all the counters contain zero and the input head is at the left of the input. The transition function maps the current state, input symbol being scanned and $k$ boolean values representing the emptiness of the counters to a new state, possibly updating any of the counters by adding or subtracting one, possibly writing an output symbol and possibly moving the input head. Counter machines can simulate Turing machines, but the best known algorithms for doing so require an exponential amount of time in general.

We now define an augmented counter machine, which extends the operations of the basic model.

Like the counter machine, an augmented counter machine (ACM) consists of a finite collection of counters and a finite state control, but the permissible operations have been expanded to include:

- Tests for equality between counters.
- Copies of contents of one counter to another.

Formally a $k$-ACM is defined like a $k$-CM except for the transition function $\delta$. In a $k$-ACM $\delta$ is:

$$\delta: Q \times I \times \{0, 1\}^{(k^2+k)/2} \rightarrow Q \times \{+1, -1, 0\}^k \times (O \cup \{\phi\}) \times H$$

where $Q$ is a set of states, $I$ is the input alphabet, $[k]$ denotes the set $\{1, 2, \ldots, k\}$, $O$ is the output alphabet and $H$ is a set of possible input head movements. The $(k^2 + k)/2$ boolean values represent the truth values of the $k$ predicates "counter $i$ contains zero" for $i = 1, 2, \ldots, k$ and the $(k^2 - k)/2$ predicates "counter $i$ is equal to counter $j$" for $j = i, i+1, \ldots, k$ for each $i = 1, 2, \ldots, k-1$. Each step can be either a "successor" step, in which counters are incremented by or decremented by one, or a "copying" step, in which each counter simultaneously receives a value from any of the $k$ counters.

It is clear that a $k$-ACM can simulate a $k$-CM in real time. For the other direction we have:

**Theorem 2.1.** Given any $k$-ACM one can effectively find a $k$-CM which linearly simulates it.

**Proof.** Given a $k$-ACM working in time $T(n)$, using the methods of [11], it should be clear how to construct an equivalent $k$-ACM $N$ which: (a) stores only non-negative integers in its counters, remembering signs in its finite state control; (b) alters at most one counter per step;
The general simulation of $N$ by $M$ thus consists of a sequence of stages corresponding to the steps of $N$. At each stage $M$ handles input and output tapes as in the corresponding step of $N$. If $N$'s step is a “successor” step, $M$'s stage will consist of a single step to update the linked list of differences. If $N$'s step is a copy step, $M$'s stage will consist of a sequence of $V$ steps, where $V$ is the number in the counter to be zeroed. The reader may verify that a transition function for $M$ can actually be effectively found given $N$'s transition function to achieve a correct simulation as above.

To complete the proof we show that $M$ simulates $N$ in linear time.

Claim. $M$ works in time $\leq (k + 1) \cdot T(n)$.

Proof of claim. A computation of $T$ steps of $N$ consists of some number $s$ of successor steps and some number $c$ of copy steps. The corresponding computation of $M$ consists of $s$ steps simulating $N$'s successor steps plus some number $C$ steps used in the $c$ stages simulating $N$'s copy steps. We will show $C \leq ks + c$. By an id of $M$ we mean a list of the contents of each of $M$'s counters. We denote by $id$ the id of $M$ immediately after stage $j$ in the simulation. Define the potential $P$ of an id as follows:

$$P(id) = \sum_{i=1}^{s} i \cdot V_i$$
where $V_i$ denotes the value in the $i$th counter (i.e. the counter with serial number $(i)$), for that id. Intuitively, the potential of an id may be thought of as the maximum cost which could be incurred in simulating any sequence of nontrivial copies starting from that id. The potential has the following properties:

- $P(id_0) = 0$.
- For all $j$, $0 \leq j \leq T$, $P(id_j) \geq 0$.
- If $id_{j+1}$ follows from $id_j$ by simulation of a successor operation then $P(id_{j+1}) \leq P(id_j) + k$.

**Proof.** Simulation of a successor operation adds one to at most one counter. If this is counter $k$ the new potential will be $k$ greater than the old; otherwise, the increase will be less.

- If $id_{j+1}$ follows from $id_j$ by simulation of a copy instruction of cost $d$ then $P(id_{j+1}) \leq P(id_j) - d + 1$.

**Proof.** If simulating the copy requires no unloading of a counter then $id_{j+1} = id_j$ and $d = 1$. Otherwise, $d$ units are unloaded from a counter and into a counter with a smaller serial number (except in the case of the last counter in M's list, which may be merely unloaded) decreasing the potential by at least $d$.

After $T = s + c$ stages consisting of time $s$ simulating successor steps and time $C$ simulating copy steps $0 \leq P(id_T) \leq k \cdot s - (C - c)$. So $C \leq k \cdot s + c$. The time taken by $M$ is therefore $s + C \leq s + k \cdot s + c = k \cdot s + t \leq (k + 1)t$. This completes the proof.

**Corollary.** A k-CM extended by the ability to determine inequality relationships{$<$, $>$} between its counters can be simulated in linear time by a k-CM.

**Proof.** In the simulation given above this information is always present in the simulating machine's finite state control. 

As Pippenger (personal communication) has pointed out, the remark on page 281 of [11] shows that a real-time simulation is impossible.

A k-tape Turing machine can simulate a k-counter machine in real-time by making each tape head's original position and then representing the contents of the CM's counters by the distances of the heads from their starting positions. Alternatively, we could use a multi-head Tm (having a single tape with $k$ heads) to do the same thing. An augmented counter machine could be real-time simulated by a multi-head TM in which heads could jump in one step to the location of any other head. Borodin (personal communication) observed that the recent result (proved independently of our Theorem 2.1) of Savitch and Vitanyi[8], which showed that multi-head Turing machines with head-to-head jumps can be simulated in linear time by multi-head Tm's without such jumps, can be improved using the methods of our proof of Theorem 2.1.

**Definition.** A k-J-Tm (k-Jump Turing machine) is a one-tape multi-head Tm which at any time instant can perform either a regular multi-head Tm move or a jump move, in which all of the $k$ tape heads are redistributed to the locations of a non-empty subset of the currently scanned tape squares.

**Definition.** A one-way infinite J-Tm is a J-Tm in which no head ever moves to the left of the starting position where they all begin.

Using standard Turing machine techniques one can show that any k-J-Tm can be simulated in real-time by a one-way-infinite k-J-Tm.

**Theorem 2.2.** Let $J$ be a one-way-infinite k-J-Tm. $J$ can be simulated in linear time by an ordinary multi-tape Tm with $4k + 1$ single-head tape units.

**Proof.** We will construct a machine $M$ which simulates $J$ in linear time using $k$ tape-units each with 2 heads (which do not jump) and one single-head tape unit. The theorem then follows by the result of Leong and Seiferas[13] which shows that the two-head tape units can be real-time simulated by four single-head units each. $M$ uses its $k$ 2-head tape units ($TU_1, \ldots, TU_k$) to keep up to $k$ tape intervals, those between each of $J$'s heads and the one
Indirect addressing and the time relationships of some models of sequential computation

between the leftmost head and the left end of J’s tape. On $TU_0$, its single head tape, $M$ keeps the interval to the right of J’s rightmost head. When J’s heads coincide one or more of M’s tape units may be free, but no more than $k$ 2-head units are ever required. $M$ keeps the heads of its 2-head units at the ends of the interval being maintained and remembers in its finite state control (1), the order in which the intervals on its units would have to be arranged to get an exact copy of J’s tape; and (2), the correspondence between J’s heads and M’s units.

This structure can be updated in one step to simulate an ordinary move of J, in which each of J’s heads may write a symbol and step left or right one square. If two of J’s heads which were coincident step apart, $M$ allocates a 2-head tape unit to maintain the new interval and adjusts its finite state control to reflect the new correspondence between J’s heads and M’s tape units. If a head of J moves right towards its neighbour, $M$ must extend the interval stored on one tape unit by one square at the right, and contract the interval stored on another by one square at the left; this can be done in one step because $M$ has heads at both ends of each interval being stored. When J’s rightmost head moves right, $M$ extends the interval stored on the corresponding tape unit by copying the symbol being scanned by its one-head tape unit and moves the head of the one-head tape unit one square to the right. Left moves by the rightmost head are handled symmetrically.

The reader may satisfy himself that such details as J’s heads crossing over one another (by ordinary moves) can be handled correctly, and that for any tape configuration and ordinary move by J, $M$ can maintain the representation described by making one move.

To describe the simulation of a jump move we first make a simplifying assumption about J: that no head is simultaneously a “jump” head and a “target” head. (In fact, this assumption doesn’t decrease the generality of the result, since by enlarging $M$’s finite state control we can always “renumber” heads in such a way that this property is maintained.) Using this assumption we can decompose a jump move by J into a sequence of $k$ single jumps, in which exactly one head jumps to the location of another.

To simulate a jump by one of J’s heads which is coincident with another head, $M$ need only adjust its finite state control to reflect the new correspondence between J’s heads and the intervals on $M$’s tape units, since these intervals do not change. To simulate a jump from a position at which no head will remain, $M$ must “coalesce” the intervals on two of its tape units, say $TU_i$ and $TU_j$, into one interval. Say $i > j$ (i.e. the physical tape unit with the lower serial number is $TU_j$). $M$ coalesces the intervals by copying, one square at a time, the contents of the squares of the interval being stored on $TU_i$ to $TU_j$, extending the interval on $TU_j$ and contracting that of $TU_i$. If $TU_i$ is storing an interval of length $d$, in $d$ steps $M$ “empties” $TU_i$ into $TU_j$ and deallocates $TU_i$, holding it in reserve for the next time a new interval is created in the simulation.

The simulation of $t$ steps of J by $M$ thus consists of $t$ stages, each following from the one before by simulation of either an ordinary step or a jump step, maintaining the storage structure described above, handling input-output just as J does. By an $id$ of $M$ we mean a list of the serial numbers of $M$’s 2-head tape units with the length $V_i$ of the interval being maintained on each. (If a tape unit $TU_i$ is not in use its interval length $V_i$ is 0.) $id_0$ represents the initial $id$, and $id_j$ represents $M$’s $id$ after $j$ stages of simulation ($1 < j < t$). The potential of an $id$, $P(id)$ is defined to be

$$P(id) = \sum_{i=1}^{k} i \cdot V_i.$$

Claim 1. $P(id_0) = 0$.

Claim 2. For all $1 \leq j \leq t$, $P(id_j) \geq 0$.

Claim 3. Say $id_{j+1}$ follows $id_j$ by simulation of an ordinary move. Then $P(id_{j+1}) < P(id_j) + k^2$.

Proof. This follows directly from the definition of potential and the facts that (1), no interval can be increased in length by more than 2; (2), the total of the lengths of all intervals increases by at most one in the simulation of an ordinary move.

Claim 3 shows that the potential rises by at most $k^2$ for each ordinary move simulated.

Claim 4 shows that the potential falls by at least the time taken to simulate a jump step-1 for each jump step simulated.
Claim 4. Say $id_{i+1}$ follows $id_i$ by simulation of a jump step of cost $d$. Then $P(id_{i+1}) \leq P(id_i) - d + 1$.

Proof. If simulating the jump does not require the "coalescing" of intervals then the potentials are the same and $d = 1$, as required. Otherwise, $d$ squares are unloaded from tape units into tape units with lower serial numbers, decreasing the potential by at least $d$.

Say in its computation of length $t$, $J$ makes $s$ ordinary moves and $c$ jump moves. Then $M$ will have $t$ stages and take $s + C$ steps where $C$ is the time spent simulating $c$ jump moves. Consider the potential at stage $t$:

$$0 \leq P(id_i) \leq s \cdot k^2 - C + c.$$ 

Thus $C \leq s \cdot k^2 + c$ and the total time taken by $M$ is

$$s + C \leq s + s \cdot k^2 + c \leq (k^2 + 1)t.$$ 

3. RANDOM ACCESS MACHINES

The random access machine (RAM), formalized by Cook[14], has been proposed as an alternative to Turing machines for the study of time complexity. Many algorithms have been described in terms of RAM programs or RAM-Algol (a high level language for RAMs developed in Reckhow[15]), partially because RAMs have many similarities to real computers, including registers, indirect addressing, assignments, arithmetic operations and branching instructions. Unlike real computers however, a RAM has an unbounded address space and each of its registers may hold an integer of any size. This has led to the definition of several types of RAM model, each differing from the others in instruction set or costing function.

The three models studied in this paper are the U-Ram (unit-cost RAM), the L-Ram (logarithmic cost RAM) and the US-Ram (unit-cost successor RAM). Each model is defined as consisting of an infinite sequence of registers $x_0, x_1, x_2, \ldots$, a read-only input tape and finite input alphabet $I$, a one-way output tape and finite output alphabet $O$, and a program, which is a finite sequence of instructions. The allowable instructions, their timings and effects for each of the three models are in Table 1. The last instruction must be a HALT instruction.

Initially all registers contain zero and the input tape head is at the left end of the input. Instructions are executed in sequence starting from the first, except as noted in Table 1. It can be seen that the U-Ram and L-Ram are distinguished by their timing functions. On the U-Ram unit time is charged for each instruction execution; whereas on the L-Ram the time taken to execute an instruction depends on the size of the contents of the registers accessed (as given by the function $l(n) = \max \{1, \log \|n\|\}$).

Like the U-Ram, the US-Ram works under the unit cost criterion, but its instruction set has been reduced by eliminating general addition and subtraction of register contents (instructions 5, 6). The only register arithmetic possible on the US-Ram is the use of successor functions (addition or subtraction of 1).

Other "macro" instructions can be implemented using the basic ones given in Table 1. For example, tests for equality between registers can be performed on a US-Ram using a fixed number of "working registers" not otherwise involved in the computation, in the following way: An instruction such as

"IF $x_j = x_k$ THEN WRITE $a$"

can be simulated using indirect addressing (Fischer[16]) by the following sequence of register transfers, after first checking to see that none of the registers $x_0, x_1, x_j$ and $x_k$ contain the constants 0, 1, $j$ or $k$.

$$x_0 \leftarrow x_j \quad x_1 \leftarrow x_k \quad x_{x_j} \leftarrow 1 \quad x_{x_k} \leftarrow 0$$
Table I. RAM instructions, effects and execution timing

<table>
<thead>
<tr>
<th>Instructions $(i,j,k \in \mathbb{N})$</th>
<th>U-Ram</th>
<th>US-Ram</th>
<th>L-Ram</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. $x_i + c$ ($c$ an integer)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2. $x_i + x_j$</td>
<td>1</td>
<td>1</td>
<td>$x_j$</td>
</tr>
<tr>
<td>3. $x_i + x_j$</td>
<td>1</td>
<td>1</td>
<td>$x_j$</td>
</tr>
<tr>
<td>4. $x_i + x_j$</td>
<td>1</td>
<td>1</td>
<td>$x_j$</td>
</tr>
<tr>
<td>5. $x_i + x_j + x_k$ (U-Ram &amp; L-Ram only)</td>
<td>1</td>
<td>N.A.</td>
<td>$x_j + x_k$</td>
</tr>
<tr>
<td>6. $x_i + x_j + x_k$ (U-Ram &amp; L-Ram only)</td>
<td>1</td>
<td>N.A.</td>
<td>$x_j + x_k$</td>
</tr>
<tr>
<td>7. $x_i + x_j + x_k$</td>
<td>1</td>
<td>1</td>
<td>$x_i$</td>
</tr>
<tr>
<td>8. $x_i + x_j + x_k$</td>
<td>1</td>
<td>1</td>
<td>$x_i$</td>
</tr>
<tr>
<td>9. $x_i + x_j + x_k$</td>
<td>1</td>
<td>1</td>
<td>$x_i$</td>
</tr>
<tr>
<td>10. GO TO n ($n \in \mathbb{N}^*$)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11. INPUT</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12. INPUTR</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13. WRITE $\sigma$ ($\sigma \in \mathbb{D}$)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>14. IF $x_j = 0$ THEN</td>
<td>1</td>
<td>1</td>
<td>$x_j$</td>
</tr>
<tr>
<td>15. IF $x_j &gt; 0$ THEN</td>
<td>1</td>
<td>1</td>
<td>$x_j$</td>
</tr>
<tr>
<td>16. IF INPUT = $\sigma$ THEN ($\sigma \in \mathbb{I}$)</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Note: $\|m\| = \text{max}(1,|\log |m||)$ where $|m|$ denotes absolute value of $m$.

Effects of Instructions:

1. $x_i$ is assigned integer $c$.
2. $x_i$ is assigned the contents of $x_j$.
3. If the contents of $x_j$ are nonnegative, $x_i$ is assigned the contents of the register whose address is found in $x_j$; otherwise, the machine halts.
4. If the contents of $x_i$ are nonnegative, the register whose address is found in $x_i$ is assigned the contents of $x_j$; otherwise the machine halts.
5. $x_i$ is assigned the sum of the contents of $x_i$ and $x_k$.
6. $x_i$ is assigned the (signed) difference between the contents of $x_j$ and $x_k$.
7. The contents of $x_j$ are incremented by one.
8. The contents of $x_j$ are decremented by one.
9. Execution is stopped.
10. The next instruction executed will be the $n$th; if no such instruction exists, HALT.
11. The input tape head is moved left.
12. The input tape head is moved right.
13. The symbol $\sigma$ is written on the output tape and its head is moved right.
14. The next instruction, which may not be an "IF" instruction, is executed iff the contents of $x_j = 0$; otherwise it is skipped.
15. The next instruction, which may not be an "IF" instruction, is executed iff the contents of $x_j$ are positive.
16. The next instruction, which may not be an "IF" instruction, is executed iff the input tape head is scanning a square with $\sigma$ on it.

---

If $x_j = 0$ THEN WRITE $\sigma$

$x_j \leftarrow x_0$

$x_k \leftarrow x_1$

where $x_0$ and $x_1$ are used as working registers.

One way in which our RAMs differ from those of Reckhow[15] is in their input conventions. Under Reckhow's definitions a U-Ram can read in an arbitrary integer $n$ in cost 1, and an L-Ram can do the same in cost $\log n$. On our models input always takes the form of strings of symbols over a finite input alphabet $I$, which can be read at most one symbol at a time. Thus for our U Ram to input an integer $n$ to a single register takes time about $\log n$; the same task can be done on a L-Ram in time $O(\log^2 n)$. (Input to L-Rams is discussed further in Section 4.)

Reckhow[15] and Cook proved that a small increase in the running time of a U-Ram or an L-Ram allows additional sets to be accepted; their result is tighter than the equivalent hierarchy theorem for Turing machines.


In Fischer[16], it is shown that the US-Ram can simulate in real-time storage modification
machines, and conversely that these machines can simulate US-Rams in real-time. Storage modification machines were developed by Schönhage[18] to model list processing computations. Schönhage showed that these machines could real-time simulate multi-dimensional Turing machines and thus, US-Rams can simulate multi-dimensional Turing machines in real time.

The U-Ram can trivially simulate the US-Ram in real time, by simply performing the same instructions.

After $t$ steps of a computation by a U-Ram $M$ the largest number stored in any register is

$$c \cdot 2^{-t}$$

where $c$ is the largest constant appearing in $M$'s program, since the contents of the largest register may at most be doubled at each step once the value $c$ is attained. An L-Ram simulating $M$ by executing exactly the same sequence of instructions will thus incur costs $O(\log (c \cdot 2^{-t})) = O(t)$ to execute the $t$th instruction. Thus the L-Ram can simulate the U-Ram in time $O(T^t)$.

In a US-Ram $M$ after $t$ steps the largest number in any register is $c + t - 1$, and so we can conclude that the L-Ram can simulate the US-Ram in time $O(T \log T)$. The following theorem appears in [16].

**Theorem 3.1.** Let $L$ be an L-Ram which works in time $T(n)$. There exists a US-Ram $M$ which linearly simulates $L$.

Thus we have a ranking of RAM models:

$$U\text{-Ram} \rightarrow US\text{-Ram} \rightarrow L\text{-Ram}$$

where $\rightarrow$ denotes a linear simulation by the model on the left of the model on the right. In the other direction, the best simulations known take more than linear time:

$$U\text{-Ram} \xleftarrow{T^{2}} US\text{-Ram} \xleftarrow{T^{\log T}} L\text{-Ram}$$

where $\xleftarrow{f}$ $M_2 \xleftarrow{f} M_1$ means that model $M_1$ can simulate model $M_2$ in time $O(f)$.

It remains an open problem to specify more closely (i.e. to within a linear factor) the inter-relationships of these three models, either by improving the nonlinear simulations or establishing that this cannot be done.

Hopcroft et al.[5], using the fact that in a small number of steps a Turing machine can make only limited changes to its tapes, proved the following theorem which relates U-Rams to Tm's:

Let $M$ be a $k$-Tm working in time $t(n) \geq n \log n$, and let $t$ be such that $\log (t(n))$ can be computed on a U-Ram in time $t(n)/\log t(n)$. Then $M$ can be simulated by some U-Ram in time $O(t(n))$.

Thus, under very general conditions, Turing machine computations can be “sped up” by U-Rams. Two corollaries follow directly from the proof of the theorem.

**Corollary 1.** Under the hypotheses of the theorem, $M$ can be linearly simulated by some L-Ram.

The proof of this corollary depends on the fact that in the course of the U-Ram simulation, no number larger than a fixed polynomial in $t$ is ever computed. Thus under the logarithmic cost criteria each instruction costs $O(\log t)$ to execute.

**Corollary 2.** The Tm cannot linearly simulate the U-Ram.

**Proof.** Assume the contrary. By the U-Ram time hierarchy theorem[15] there exists a U-Ram $M$ working in some constructible monotonic time bound $T(n) \approx n \log n$ accepting a set $S$; and furthermore no U-Ram working in any time $T_t$ such that $\lim_{n \rightarrow \infty} (T_t(n)/T(n)) = 0$ accepts $S$.

By assumption there exists a Tm $M'$ which accepts $S$ and works in time $cT$. Applying the
theorem we obtain a U-Ram $M_0$ which accepts $S$ in time $O(cT/\log T)$; but this is a contradiction, since

$$\lim_{n \to \infty} \frac{O\left(\frac{cT(n)}{\log T(n)}\right)}{T(n)} = 0.$$ 

The techniques of the theorem suggest a method for speeding up counter machine computations by a U-Ram. As the next theorem shows, in this case the acceleration is by a polynomial factor depending on the number of counters.

**Theorem 3.2.** Let $T$ be such that $T(n) \geq n$ and $(T(n)/n)^{1/(k+2)}$ is constructible on a U-Ram. Then for any $k$-CM $C$ which works in time $T$ we can effectively find a U-Ram $M$ which simulates $C$ and works in time

$$O(n^{1/(k+2)} \cdot T^{(k+1)/(k+2)}) = O\left(T\left(\frac{T}{n}\right)^{1/(k+2)}\right).$$

**Proof.** Assume $C$ has $m$ states and that $C$ stores only non-negative integers in its $k$ counters. $M$'s simulation of $C$ will take the form of a sequence of stages, each stage simulating many consecutive steps of $C$. $M$ begins by determining $n$ and calculating $(T(n)/n)^{1/(k+2)} = q$.

In $q$ steps of $C$'s computation starting from any configuration, no counter can have its value altered by more than $q$ (since at each step the value can change by at most 1). Moreover, any counter $R$ whose initial value $v$ is $q$ or greater has a nonzero value throughout the $q$ steps. Had the contents of $R$ instead been some greater value $v + i$ ($i > 0$) at the start, the same sequence of $q$ transitions would be taken by $C$ since in either case $R$ would have nonzero contents throughout. In addition, if after $q$ steps the final value in $R$ is $v + d$ for some offset $d$ from its original contents, then had $R$ began instead with contents $v + i$, its final contents would have been $v + i + d$ (in either case the same sequence of increment-decrement operations would be applied).

The fact that values $\geq q$ can be treated alike in simulating $q$ steps of $C$ motivates the following procedure: $M$ first precomputes the outcomes of $q$ steps of $C$ starting from each of many possible configurations, storing the results in a table; and then can simulate blocks of $q$ steps of $C$ by simple table lookup operations, taking a small fixed number of steps for each lookup.

More precisely, $M$'s simulation will consist of an initialization phase followed by a simulation phase in which $T$ steps of $C$ will be simulated in $T/q$ stages.

In the simulation phase $M$ keeps in $k$ registers $x_1, x_2, \ldots, x_k$ the contents $v_1, v_2, \ldots, v_k$ of the counters of $C$, and in two additional registers $s_{k+1}, s_{k+2}$ two integers which represent the counter machine’s state $s$ and position of the input tape head $h$. To simulate $q$ steps of $C$, $M$ forms a vector $\langle s, h, \bar{v}_1, \bar{v}_2, \ldots, \bar{v}_q \rangle$ where $\bar{v}_i = \min(v_i, q)$. Using this vector as an index into a $k + 2$-dimensional table $M$ finds a resultant vector $\langle s’, h’, d_1, d_2, \ldots, d_k \rangle$ in which $s’$ and $h’$ are the resultant state and head positions and $d_i$ is an integer giving the change in counter $i$’s value after the $q$ steps. Using this resultant vector $M$ updates $x_1, \ldots, x_{k+2}$. Thus in time $O(k) M$ simulates $q$ steps of $C$ and the total time spent in the simulation phase is therefore $O(k \cdot (T/q))$.

In the precomputation phase $M$ reads the input, storing it in $n$ consecutive registers. For each possible $k + 2$-tuple $M$ first computes the resultant vector by simulating $q$ steps of $C$ directly then stores it in the $k + 2$ dimensional table. (The details of the “linear time” implementation of such a table are discussed later.) There are $m \times n \times (q+1)^k$ such tuples, each of which must be simulated for $q$ steps, then added to the table, and so the time needed for the precomputation is $n + O(m \times n \times q^k \cdot (q+1)) = O(nq^{k+1})$.

The total time taken in the entire simulation is therefore

$$O\left(k \cdot \frac{T}{q}\right) + O(n \cdot q^{k+1}) = O\left(k \cdot \frac{T}{(T/n)^{1/(k+2)}}\right) + O\left(n \cdot \left(\frac{T}{n}\right)^{(k+1)/(k+2)}\right) = O(n^{1/(k+2)} \cdot T^{(k+1)/(k+2)})$$

\(\square\)
Corollary 1. Under the hypotheses of Theorem 3.2, there exists an L-Ram simulating C in time
\[ O(n^{(k+2)} \cdot T^{-(k+1)/(k+2)} \log T). \]

Proof. The largest number appearing in the U-Ram simulation is \( O(T^3) \). Thus an L-Ram executing the same program incurs costs \( O(\log T) \) at each step.

Corollary 2. For any time bound \( T, T(n) \geq n \), there exist sets accepted by an L-Ram in time \( T \) which cannot be accepted by any CM working in time \( O(T) \).

Corollary 3. For any time bound \( T, T(n) \geq n \), there exist sets accepted by a US-Ram in time \( T \) which cannot be accepted by any CM working in time \( O(T) \).

Proof. This follows from the previous corollary and the linear simulation of the L-Ram by the US-Ram.

In fact, a stronger statement can be made. The speedup by a fixed polynomial shows that for any time bound \( T, T(n) \geq n \), there exist sets which can be accepted in time \( T \) by an L-Ram, but which cannot be accepted by any CM working in time \( T \log^a T \) for any \( a \in \mathbb{N} \).

We can also apply the speedup technique to multi-dimensional Turing machines. For \( d, k \geq 1 \) a \( d \times D k \times Tm \) is a \( d \)-dimensional Turing machine in which \( k \) \( d \)-dimensional storage spaces replace the \( k \) (1 dimensional) storage tapes; at any step in its computation depending on its current state, the current input symbol and the symbols being scanned by its workspace heads, \( D \) may change state, possibly write a symbol on its output tape and move its input head one square, write new symbols on each of the \( k \) squares currently scanned by its \( k \) tape heads, and move these tape heads, each no more than one square positively or negatively along any of the \( d \) orthogonal dimensions. Initially all squares in the workspaces are blank. The transition function of such a machine is a map

\[ \delta: Q \times I \times \Sigma^k \rightarrow Q \times (O \cup \{e\}) \times \{L, R, 0\} \times \Sigma^k \times ((\phi) \cup ([d] \times \{+, -, 0\}))^k \]

where \( Q \) is a finite set of states, \( I \) is the input tape alphabet, \( \Sigma \) is the storage space alphabet, \( O \) is the output alphabet, \( \{L, R, 0\} \) represent input tape head movement instructions, and \( [d] \times \{+, -, 0\} \) represents workspace head movement instructions along any of \( d \) dimensions.

In the proof to be given it will clarify the exposition to make some simplifications to the \( d \times D k \times Tm \) which involve no significant loss of generality. By a simple \( d \times D k \times Tm \) we mean one in which the workspace heads never leave the subspaces defined by the origin and the positive directions in each of the \( d \)-dimensions and one which, in its first \( n \) moves, copies its input onto \( n \) successive squares of the first dimension of the first workspace never again moving its input tape head. It should be clear that given any \( d \times D k \times Tm \) \( D \) we can find a \( d \times D k + 1 \times Tm \) which linearly simulates \( D \).

Theorem 3.3 Let \( D \) be a simple \( d \times D k \times Tm \) which works in time \( t, t(k(n) \geq n \log n \), and let \( t/(\log t)^{1/d} \) be constructible on a U-Ram. Then there exists a U-Ram \( M \) which simulates \( D \) in time \( O(t/(\log t)^{1/d}) \).

Proof. We will illustrate the construction for the case of a two symbol tape alphabet and \( q \) states. Let \( c = 2 \cdot k \cdot 3^d \).

A computation by \( D \) of \( t \) steps uses at most \( t \) tape squares in each workspace and is contained entirely in a \( d \)-dimensional hypercube of \( t^d \) tape squares for each workspace. Consider this hypercube divided into blocks, each a \( d \)-dimensional hypercube with side of length \( s = (\log t/c)^{1/d} \). Since each square in the block has one of the two workspace symbols written on it, the total number of possible different blocks is \( 2^{\log c} = t^{1/c} \). By interpreting the contents of the tape squares as a binary representation of an integer \( i \) in some standard way we can associate with each block its corresponding serial number \( i \). (A serial number of 0 will correspond to a block all of whose squares are blank.)
Indirect addressing and the time relationships of some models of sequential computation

Now consider that in \( s \) steps of \( D \) only the \( k \) blocks in which the heads were initially present and possibly their \( k \cdot (3^d - 1) \) immediately adjacent blocks can be altered. We call this set of \( k \cdot 3^d \) blocks the block configuration. In fact, these \( k \cdot 3^d \) blocks, their head positions and initial state completely determine the next \( s \) steps of \( D \)'s computation. This, and the limited number of possible distinct blocks, suggest that instead of simulating each step of \( D \) we could precompute the result of taking \( s \) steps from a given block configuration and then simply "lookup" this result whenever this configuration is encountered. If the lookup can be done quickly, i.e. in some constant number of steps, this procedure will lead to a computation of \( O(t/s) \) steps to simulate \( t \) steps of \( D \).

\( M \)'s simulation of \( D \) will therefore consist of three phases: an initialization phase, a precomputation phase and a simulation phase.

The simulation phase

After any multiple of \( s \) steps the machine is in some state \( q' \) and for each \( 1 \leq i \leq k \) \( D \)'s workspace \( i \) has a head in a particular block \( b_i = b_{ij_1,j_2,\ldots,j_d} \) where for \( 1 \leq i \leq d \), \( j_i \) describes the number of blocks between \( b_i \) and the \( l \)th axis (\( 0 \leq j_i \leq (ds) \)). The neighbours of this block are the \( 3^d - 1 \) blocks \( b_{ij_1,j_2,\ldots,j_d} \) as each \( j_i \) ranges from \( j_i - 1 \) to \( j_i + 1 \). The total configuration of \( D \) is represented in the U-Ram \( M \) in this way:

- \( x_0 \) contains an integer between 1 and \( q \) representing \( q' \), the current state.
- For each workspace \( i \) \( M \) uses \( (ts)^d \) consecutive registers each containing the serial number of the corresponding block in \( M \), arranged in increasing order of subscripts interpreted as a base \( (ts) \) number. We leave it to the reader to satisfy himself that given this representation and the address of the register representing a block, the addresses of the registers representing the \( 3^d - 1 \) neighbouring blocks may be easily obtained by adding and subtracting (precomputed) powers of \( ts \).
- Also for each of the \( k \) workspaces, \( M \) uses 2 additional registers, one giving the address of the register representing the block which contains the head, the other specifying the position of the head within this block by an integer \( p_i \), \( 1 \leq p_i \leq (\log t/c) \).

To update this representation to reflect \( s \) more steps of the Turing machine, \( M \) uses a precomputed \( 1 + k \cdot (3^d + 1) \) dimensional table in which the tuple \( v \) (consisting of (the integer representing) the current state and for each \( 1 \leq i \leq k \), the serial numbers of the block with the head in it and its \( 3^d - 1 \) neighbours and \( p_i \), the position of the head in its block) is mapped to a resultant tuple which describes the changes which must be made to \( M \)'s representation to reflect \( s \) more steps of \( D \) (i.e. the new state and, for each of the \( k \) workspaces, the new serial numbers for the blocks and new head positions \( p_i \) as well as instructions for updating the register pointing to the block containing the head if the head has moved to a neighbouring block).

The table can be implemented so that access time is \( O(1 + k \cdot (3^d + 1)) \) and thus the entire time required to form the tuple, use it as an index into the table and perform the update is \( O(k \cdot 3^d) \). In this way the time required for the simulation phase will be \( O((ts) \cdot k \cdot 3^d) = O(t/s) \) since \( k \) and \( d \) are fixed. Since \( s = (\log t/c)^{1/d} \), the simulation phase requires \( O(t/(\log t)^{1/d}) \) steps.

The precomputation and initialization phases

In the precomputation phase \( M \) must initialize the \( 1 + k(3^d + 1) \) dimensional table described above. The number of entries to be made is the number of distinct \( 1 + k(3^d + 1) \) tuples, which is \( q \times (s^d \times (ts)^d)^k \) and to compute the resultant updating information requires a direct simulation of \( s \) steps of \( D \). The time required to initialize such a table is linear in the number of entries so that the total time spent in precomputation is

\[
P = O(q \times (s^d \times t^{3d/c})^k \times (s + k \cdot 3^d)) = O(s^{kd+1} \times t^{kd/c})
\]

since \( k \), \( d \), \( q \) and \( 3 \) are fixed.
Having chosen $c = 2 \cdot k \cdot 3^d$ and $s = (\log t/c)^{1/d}$

$$P = O\left(\frac{(\log t)^{(kd+1)/d}}{c^{(kd+1)/d}} \times t^{1/2}\right) = O\left(\log^{k+1} t \cdot \sqrt{t}\right).$$

In the initialization phase, the input must be read in and converted to block serial numbers so that the initial representation of $D$ will be correct; $s, t/s$ and the numbers $(t/s)^2, (t/s)^3, \ldots, (t/s)^d$ must be calculated. This can be done in time $O(t/\log t)$.

The total time for all 3 phases of the simulation is thus dominated by the time for the simulation phase and the theorem follows. \hfill \Box

As an immediate corollary of this theorem and the U-Ram hierarchy theorem we observe that for every time bound $T$, $T(n) \equiv n \log n$, there are sets which can be accepted by a U-Ram in time $T$ but which cannot be accepted by any multi-dimensional Turing machine in time $cT$ for any constant $c$.

The proofs of Theorems 3.2 and 3.3 rely on the ability of the U-Ram to efficiently implement multi-dimensional arrays so that successive configurations of the computation can be obtained quickly. Using indirect addressing and then adding to compute offsets allows any entry of a (pre-existing) $k$-dimensional array to be accessed in $2 \cdot k$ steps of the U-Ram. The array can be viewed as a tree of $k + 1$ levels in which leaf nodes represent array entries, the first $k$ levels correspond to the $k$ dimensions of the array and the branching factor of all nodes at level $i$ is equal to the extent $b_i$ of the corresponding dimension. In the U-Ram implementation of the tree, registers represent nodes and all sons of a node are stored in consecutive registers. Each father node holds the address of his first son; thus the address of any of his sons can be quickly calculated. The size of the tree is $1 + \sum_{i=1}^{k} \prod_{j=1}^{i} b_j$ and the number of leaves is $\prod_{j=1}^{k} b_j$. It can be shown by induction that the number of internal (non-leaf) nodes is less than the number of leaf nodes if each dimension has extent $\geq 2$. The time to set up this tree structure is then $O(\prod_{i=1}^{k} b_i)$, which is linear in the number of array entries.

Although we have thus far been unable to prove that the additional instructions of the U-Ram allow it to accept more sets than a US-Ram in a given time bound, Rackoff (personal communication) conjectured that the specific programming technique of directly implementing $d$-dimensional arrays (such that any entry could be accessed in $d$ steps given $d$ registers containing "subscripts") is not possible on US-Rams. To make this idea precise we consider the case of a $n \times n$ 2-dimensional square array. By a solution to the 2-D array access problem of size $n$ we mean a triple $(R, f, C)$ where $R$ is a US-Ram, $f$ is an injective function $f: [n]^2 \rightarrow N$ and $C$ specifies the initial contents of $R$'s registers $x_2, x_3, x_4, x_5, \ldots$, such that when started with two integers $v_0$ and $v_1$ $(1 \leq v_0 \leq n, 1 \leq v_1 \leq n)$ placed in registers $x_0$ and $x_1$, $R$ computes $f(v_0, v_1)$ and stores its value in $x_3$. We interpret this statement of the problem as follows:

---For a given $n$, $f$ is a one-one map from each possible pair of array subscripts to the address of the register containing the corresponding array element.

---The US-Ram $R$ can initially contain in its registers $x_2, x_3, \ldots$ any information depending on $n$ but not depending on the specific input subscripts $v_1, v_2$.

---$R$ computes $f(v_0, v_1)$ and stores it in $x_3$.

**Theorem 3.4.** For every constant $d$, for all sufficiently large $n$, any solution $(R, f, C)$ to the 2-D array access problem of size $n$ is such that $R$ requires more than $d$ steps on some input pair $(v_0, v_1)$.

**Note:** It should be pointed out that while this theorem gives a lower bound on a specific programming technique in a very strong way (by allowing a different technique for every $n$) it says nothing about differences in the time-bounded acceptance powers of the U-Ram and US-Ram.

**Proof.** Suppose the contrary. Then there exists a constant $d$ such that for arbitrarily large $n$ there is a solution $(R, f, C)$ which requires $\leq d$ steps on all input pairs $(v_0, v_1)$. The computations of $R$ on all $n^2$ possible input pairs can be described by means of a rooted directed tree in which
Indirect addressing and the time relationships of some models of sequential computation

The sequence of instructions labelling the branch $b$ have certain effects on the storage of $R$ which may depend on the specific input pairs. We wish to analyze the total number of different possible values which could be placed in $x_3$ by all the possible input pairs for which this branch is followed. To do this we will use the concept of a (potentially) altered register, that is a register which could have been altered by execution of the instructions on $b$; and we will define the set $V_A(i)$ to be the set of all the values which could occur in registers which are potentially altered after $i$ instructions have been executed. By convention, $x_0$ and $x_1$ are viewed as altered registers, and since they each may initially contain $n$ different values, $|V_A(0)| = n$.

Claim. For all $i$, $|V_A(i)| \leq n \cdot 2^i$. That is the total number of possible values in the registers which could have been altered at most doubles for each instruction executed.

Proof (by induction on $i$). The base step $i = 0$ follows from the definitions. For the induction step we show that execution of the $i + 1$th instruction can at most double the size of $V_A$ (i.e. $|V_A(i + 1)| \leq 2 \cdot |V_A(i)|$) by considering each possible type of instruction.

Case 1. Input-output, testing and branching instructions do not alter registers and thus do not affect $V_A$.

Case 2. ($i + 1$th instruction is of the form $x_j \leftarrow x_k$). If $x_k$ has been altered $V_A(i + 1) = V_A(i)$. If not, $|V_A(i + 1)| = |V_A(i)| + 1$ since only the value in $x_k$ is added to the set $V_A(i)$.

Case 3. ($x_i \leftarrow x_j + 1$). If $x_i$ has been altered, it could contain any of the values in $V_A(i)$. Adding 1 to reach of these values gives us at most $|V_A(i)|$ possible new values. If $x_i$ has not yet been altered then at most one element is added to $V_A(i)$.

Case 4. ($x_i \leftarrow x_j - 1$). As in case 3.

Case 5. ($x_i \leftarrow x_j$). If $x_j$ has been altered, no new elements are added to $V_A(i)$; otherwise, one new value is added.

Case 6. ($x_i \leftarrow x_k$). If $x_k$ is unaltered this is equivalent to case 3; one new value may be added. If $x_j$ has been altered it can take on at most $|V_A(i)|$ different values and the registers indexed by these values may all contain new values. Thus $|V_A(i + 1)| \leq 2 \cdot |V_A(i)|$.

This completes the proof of the claim. Thus after $d$ steps over all the input pairs which cause this branch to be selected (there are at least $n^2/2^d$), no more than $n \cdot 2^d$ different values could be placed in any of $R$'s registers, and in particular, in register $x_3$. Observing that $n \cdot 2^d < (n^2/2^d)$ wherever $n > 2^{2d}$ shows that this branch cannot work correctly for $n^2/2^d$ different input pairs, contradicting the assumption and completing the proof.

The details of the proof of the theorem establish worst and average case lower bounds of $\Omega(\log n)$ steps for an access; this bound ignores the kind of tests used to select the branches. If the instruction set of the US-Ram were extended by permitting comparisons of register contents (such as "IF $x_i > x_j$ THEN") the above theorem would still hold. Rackoff has since observed that on the US-Ram for all $n \geq 10(\log n)$ steps (with no tests of any kind) are sufficient to solve the 2-D array access problem. Thus the above lower bound is optimal to within a constant.

A 2D-US-Ram could be defined to be a RAM on which an unbounded plane of registers were available and program register references were doubly subscripted. On such a model, not only the 2-D array access problem but the analogous problem for higher dimensions could be solved quickly (i.e. in a number of steps depending only on the number of dimensions).

4. CONCLUSIONS AND OPEN PROBLEMS

Other types of RAM model have been proposed, varying from the ones presented here in instruction set or cost function. One significant change is the introduction of parallelism. For example, a model proposed by Goldschlager[19] has an instruction set similar to our L-Ram, but the timings of some instructions have been changed to reflect more closely the costs which would be incurred in a physical realization of the model using highly parallel circuitry. (For
example, the cost of an indirect assignment instruction \( x_i \leftarrow x_j \) would be \( l(x_i) + \log l(x_j) \) in contrast to the L-Ram's cost of \( l(x_i) + l(x_j) \). Stockmeyer [20] has studied the relation between the U-Ram and the more powerful vector machine, in which the arithmetic operations of the U-Ram have been replaced by bit-wise boolean operations between registers and shift instructions for individual registers. We have not here considered such parallel models, restricting our attention to sequential machines.

In Aho, Hopcroft and Ullman [21] many algorithms are described in terms of RAM programs in which instructions can include the computation of products, the "mod" function, quotients and many other functions. Whether these algorithms could all be implemented on a U-Ram to work in the same time bound (to within a linear factor) is an interesting open problem.

Kasai [22] has studied the relationship between the L-Ram and a similar model in which no indirect addressing is permitted, as well as showing that for on-line computations the \( T^2 \) simulation of the L-Ram by the TM (Reckhow [15]) cannot be improved.

It is interesting to note that while we have a linear simulation of Turing machines by L-Rams, this result applies only for Tm's working in time \( t(n) \geq n \log n \). This suggests the possibility that L-Rams may not be able to linearly simulate linear time-bounded Turing machines. With an L Ram can accept the set \( L_1 = \{a^n b^n | n \geq 1\} \) in linear time (by using separate registers to hold each bit of a "counter" with lower order bits stored in lower numbered registers), we conjecture that there are sets (such as \( L_2 = \{x \in \{0, 1\}^* | x^R \) denotes the reversal of the string \( x \}) which cannot be accepted by any L-Ram in linear time. What appears to make \( L_2 \) more difficult to accept on the L-Ram is the fact that, from an information theoretic viewpoint to accept a word of length \( n \) belonging to \( L_1 \) in linear time only \( O(\log n) \) bits of information need be stored, while \( L_2 \) may require \( \Omega(n) \) bits to be stored for linear time acceptance. Pippenger has observed that the L-Ram can load \( k \) input bits into a register in time \( O(k^2) \), and thus could store away a binary input of length \( n \) in cost \( O(n \cdot (\log n)^{1/2}) \) by storing \( (\log n)^{1/2} \) bits to a register. It remains an open problem to establish a nonlinear lower bound for this problem.

It would also be of interest to develop tighter bounds for the simulation of the US-Ram by the L-Ram. Borodin has conjectured that given any initial L-Ram configuration, the number of different configurations which could be reached via any computation of cost \( q \) is \( O(2^{q \sqrt{\log q}}) \). In contrast to this, a US-Ram with appropriate initial contents can reach \( \Omega(2^q) \) distinct configurations via computations of \( t \) steps. Thus a proof of Borodin's conjecture might lead to a proof that the L-Ram cannot linearly simulate the US-Ram.

Observing the various speedups of other models possible on the U-Ram, another open problem suggests itself: under what conditions can any uniform Bounded Activity Machine with polynomially limited accessibility (in the sense of Cook and Aanderaa [23]) be simulated by a U-Ram (1), in linear time? or (2), with speedup?

A US-Ram whose program contains no indirect addressing instructions is essentially an augmented counter machine, since both models perform the same kinds of tests and operations on storage. The sets accepted by polynomial time bounded counter machines are all members of DLOG = \( \{A | A \) is a language accepted by a log space bounded Turing machine\}. The sets accepted by polynomial time bounded US-Rams comprise exactly \( P = \{A | A \) is a language accepted by a polynomial time bounded Turing machine\}. It is a long-standing problem of computer science to determine the exact relationship between the sets \( P \) and DLOG. (It is generally believed that \( P \) properly contains DLOG.) The speedup of ACM's by US-Rams gives us a setting in which we can at least conclude that "indirect addressing helps" in terms of time bounded computations.

Acknowledgements—This research was done while I was a graduate student under the supervision of Allan Borodin. I'd like to thank Allan Borodin, Charles Rackoff, Nicholas Pippenger, Martin Tompa and Les Goldschlager for their help, and the National Research Council of Canada for financial support.

REFERENCES

Indirect addressing and the time relationships of some models of sequential computation