Acta Informatica (2012) 49:485–491
DOI 10.1007/s00236-012-0171-1

ORIGINAL ARTICLE

Does indirect addressing matter?

A note

Michael Brand

Received: 4 May 2012 / Accepted: 19 September 2012 / Published online: 27 October 2012

Abstract In the study of random access machines (RAMs) and the complexities associated with their algorithms, the availability of indirect addressing often creates an analysis obstacle. We show that for RAMs equipped with a sufficiently rich set of basic operations, indirect addressing does not increase computational power, and can be simulated either in linear time or on-line in real time. These results pertain to the uniform cost model and, particularly, assume a unit cost variable shift.

1 Introduction

The Turing machine, first introduced in [10], is undoubtedly the most familiar computational model. However, for algorithm analysis it often fails to adequately represent real-life complexities, for which reason the random access machine (RAM), closely resembling the intuitive notion of an idealized computer, has become the common choice in algorithm design. Quoting [3]:

> The random access machine (RAM) seems to be the computational model in widest use for algorithm design and analysis. The RAM is intended to model what we are used to in conventional programming, idealized in order to be better accessible for theoretical study. [It is] the standard platform for the study of algorithms.

A full description of RAMs is given in [1]. We summarize here briefly.

Computations on RAMs are described by *programs*. RAM programs are sets of *commands*, each given a *label*. Without loss of generality, labels are taken to be consecutive integers. The bulk of RAM commands belong to one of two types. One type is an *assignment*. It is described by a triplet containing a k-ary operation, k operands and a target register. The other type is a *comparison*. It is given two operands and a comparison operation, and is equipped with labels

M. Brand (✉)
Faculty of IT, Monash University, Clayton, VIC 3800, Australia
e-mail: michael.brand@alumni.weizmann.ac.il

to proceed to if the comparison is evaluated as either true or false. Other command-types include unconditional jumps and execution halt commands.

The execution model for RAM programs is as follows. The RAM is considered to have access to an infinite set of registers, each marked by a non-negative integer. The input to the program is given as the initial state of the first registers. The rest of the registers are initialized to 0. Program execution begins with the command labeled 1 and proceeds sequentially, except in comparisons (where execution proceeds according to the result of the comparison) and in jumps. When executing assignments, the k-ary operator is evaluated based on the values of the k operands and the result is placed in the target register. The output of the program is the state of the first registers at program termination. (Alternatively, and more resembling Turing machines, RAMs may be equipped with special halting instructions that either "accept" or do not.) For simplicity, this paper assumes that the RAM input is contained strictly in the first K registers, where K is a program-dependent constant, and that the output is read strictly from the first register. We also assume, following e.g. [5], that all explicit constants used as operands in the RAM program belong to the set $\{0, 1\}$. Neither of these assumptions makes a material difference to the results, but they simplify the presentation.

In this paper we deal with RAMs that use non-negative integers as their register contents. This is by far the most common choice. A RAM will be indicated by RAM[op], where op is the set of basic operations supported by the RAM. These basic operations are assumed to execute in O(1) time. This is known as the "uniform cost" or "unit cost" model, and entails the ability to manipulate extremely large numbers at unit time cost. Space requirements for a RAM are measured by the highest-numbered register used.

Note that because registers only store non-negative integers, such operations as subtraction cannot be supported without tweaking. The customary solution is to replace subtraction by "natural subtraction", denoted "$\dot{-}$" and defined by $a \dot{-} b \stackrel{\text{def}}{=} \max(a - b, 0)$. We note that if the comparison operator "$\leq$" (testing whether the first operand is less than or equal to the second operand) is not supported by the RAM directly, the comparison "$a \leq b$" can be simulated by the equivalent equality test "$a \dot{-} b = 0$". Testing for equality is always assumed to be supported.

Operands to each operation can be explicit integer constants, the contents of explicitly named registers or the contents of registers whose numbers are specified by other registers. This last mode is known as "indirect addressing". We note that the target register of RAM assignments may likewise be specified either explicitly or by indirect addressing.

All RAMs have indirect addressing and an infinite number of registers. However, this feature of RAMs often introduces great difficulties for analysis, which is why some papers (e.g., [8,4]) only analyze RAMs that make no use of indirect addressing (and therefore necessarily utilize only a bounded number of registers). We denote the RAM model without indirect addressing as RAM_0. Regarding this choice, the authors of [4] write

> The power of indirect addressing has not been characterized, and it is not known whether it is a substantial advantage.

This statement is largely still true. Results with and without indirect addressing are typically handled in separate papers (e.g., [8] vs. [7]), with indirect addressing requiring more advanced mathematical techniques. In [2], sufficient conditions are given, under which indirect addressing is essential, in the sense that a machine without indirect addressing cannot simulate a machine with indirect addressing. Even in cases where indirect addressing can be simulated, it may entail a performance cost. For example, in [6] a computational model is described under which the symbol table problem is $\Omega(n \log n)$ time if and only if indirect

addressing is not allowed. Furthermore, [6] proves that given indirect addressing, any program that achieves $o(n \log n)$ time performance for this task must necessarily use space unbounded by *any* function of n, the size of the input. (The symbol table problem is the problem of finding, given $(x_1, \ldots, x_n, y_1, \ldots, y_n)$ as input, for each $i : 1 \leq i \leq n$, a value, t, such that $y_t = x_i$, or asserting than no such t exists.)

We show, however, that for powerful enough operation sets, indirect addressing entails no advantage. To clarify the term "powerful": in the case of RAMs in the unit cost model many innocuous-looking operation sets are surprisingly powerful. For example, a RAM$[+, \dot{-}, \times, \div]$ can simulate any PSPACE Turing machine in polynomial time [7]. The operation sets we consider are $\{+, \dot{-}, \leftarrow, \rightarrow, \wedge\}$ (shown in [8] to also span PSPACE in polynomial time), $\{+, \dot{-}, \leftarrow, \rightarrow\}$ (not known to be as strong as the previous examples) and $\{+, \dot{-}, /, \leftarrow, \rightarrow, \wedge\}$ (shown in [9] to be even more powerful than the above).

Here $\leftarrow, \rightarrow$ and $\wedge$ denote bitwise left-shifting, bitwise right-shifting and bitwise conjunction, respectively. The symbol "/" denotes exact division, which is integer division that assumes the result has no remainder and is otherwise undefined. We use "$\div$" to indicate general integer division, which is conjectured in [9] to be more powerful than exact division.

One of the most general statements proven regarding the power of indirect addressing is given by [3]:

Theorem 1 [3] *For the set of operations $op = \{+, \dot{-}, \leftarrow, \rightarrow, \wedge\}$, a RAM$[op]$ algorithm that takes t time and s space can be simulated by a RAM$_0[op]$ in $t\alpha(s)$ time, where α is the inverse Ackermann function.*

It is conjectured in [3] that this is best possible. Note, however, that [3] works in a different model than that of the above-cited examples. It works in the model of the *on-line RAM*, which is a model where the input is not given in advance, nor is the output read at the end of the process. Rather, the RAM is equipped with "READ" and "WRITE" instructions that can be accessed at any time. The initial state of the registers is assumed to be all zeroes. The on-line model is a stronger model in the sense that the on-line model is able to simulate off-line behavior, while the converse is not necessarily true.

We complement the result of [3], by showing that for the standard (not on-line) RAM indirect addressing can be simulated at no cost to the run-time complexity of an algorithm, even using a smaller instruction set than the one used by [3]. Using a slightly larger instruction set (adding the exact division operation, "/"), we demonstrate that real-time simulation is possible even in the on-line model.

Specifically, we claim

Theorem 2 *For the set of operations $op = \{+, \dot{-}, \leftarrow, \rightarrow\}$, a RAM$[op]$ can be simulated by a RAM$_0[op]$ in linear time. (That is, a RAM$[op]$ requiring n steps can be simulated in $O(n)$ steps on a RAM$_0[op]$.)*

Theorem 3 *For the set of operations $op = \{+, \dot{-}, /, \leftarrow, \rightarrow, \wedge\}$, an on-line RAM$[op]$ can be simulated by an on-line RAM$_0[op]$ in real time. (That is, each operation of the on-line RAM$[op]$ can be simulated in $O(1)$ steps on an on-line RAM$_0[op]$.)*

2 Linear time simulation

In order to simulate a RAM on a RAM$_0$, we must first describe a method to encode the data from n registers (an unbounded number of registers) in a bounded number of registers. We encode the values $k_0, \ldots, k_{n-1}$ from n registers as follows:

- Let $Z = 2^m$ be a number greater than $\max(k_i)$.
- Let $V = \sum_{i=0}^{n-1} Z^i k_i$.

The triplet (m, V, n) contains all the information of the original values. We use the following terminology.

Definition 1 A triplet (m, V, n) of integers will be called an *encoded vector*. We refer to m as the $width$ of the vector, V as the *contents* of the vector and n as the $length$ of the vector. If $V = \sum_{i=0}^{n-1} 2^{mi} k_i$ with $\forall i : 0 \leq i < n \Rightarrow 0 \leq k_i < 2^m$, then $[k_0, \ldots, k_{n-1}]$ will be called the *vector* (or, the *decoded vector*), and the k_i will be termed the *vector elements*. Notably, vector elements belong to a finite set of size 2^m and are not general integers. It is well-defined to consider the most-significant bits (MSBs) of vector elements. Nevertheless, any n integers can be encoded as a vector, by choosing a large enough m.

Actions described as operating on the vector are mathematical operations on the encoded vector (typically, on the vector contents, V). However, many times we will be more interested in analyzing these mathematical operations in terms of the effects they have on the vector elements. Where this is not ambiguous, we will name vectors by their contents. For example, we can talk about the "decoded V" to denote the decoded vector corresponding to some encoded vector whose contents are V.

We begin by considering a simpler scenario than that of Theorem 2.

Lemma 1 *For the set of operations $op = \{+, \dot{-}, \leftarrow, \rightarrow\}$, the first n execution steps of a RAM[op] can be simulated in $O(n)$ time by a RAM$_0$[op] program to which n is given as an additional input.*

Proof Let $M = M_{op}(n, input)$ be the largest number that can appear in any register of a RAM[op] initialized by $input$ in the course of its first n execution steps. We take $input$, in this context, to be a K-tuple. We denote its i'th element by $input_i$.

Our first claim is that there is a RAM$_0$[op] initialized by $input$ and n that calculates $M_{op}(n, input)$ in O(n) execution steps. For our particular choice of op, an example of such a RAM$_0$ is one that first calculates

$$r_1 \Leftarrow \max\left(\{input_i : 0 \leq i < K\} \cup \{1\}\right),$$

and then goes through n rounds of

$$r_{i+1} \Leftarrow r_i \leftarrow r_i.$$

Here, "$\Leftarrow$" indicates assignment, as opposed to "$\leftarrow$" which indicates left shifting.

Let us temporarily assume that the RAM$_0$ supports multiplication. We want to store all registers of the original RAM as an encoded vector of width large enough to store M. One way to do this is to choose $m = M$. It takes O(n) steps to calculate this m value. Following this, loading the value stored in register i, for any i, can be simulated by

$$\text{LOAD}(i) = (V \rightarrow mi) \dot{-} ((V \rightarrow (mi + m)) \leftarrow m),$$

and storing a new value N in register i can be simulated by

$$V \Leftarrow V + (N \leftarrow mi) \dot{-} (\text{LOAD}(i) \leftarrow mi).$$

Hence, each instruction can be simulated in O(1) time, following an O(n)-time setup, for a total of O(n) operations.

We note that the only use made of multiplication is in calculating mi. Therefore, by switching to a vector of width $1 \leftarrow m$ rather than width m, this multiplication can be made into a left shift. □

We now move to the general case, where n is not available as part of the input.

Proof (of Theorem 2) The proof of Lemma 1 allows us, in $O(n)$ time, to execute a $RAM_0[op]$ program $simulate(n)$ that simulates the first n execution steps of the target RAM. Without expanding the operation set, we can run on the same RAM_0 the program

```
1: n ⇐ 1
2: loop
3:    simulate(n)
4:    if Simulation halts then
5:       return Halting state
6:    end if
7:    n ⇐ n + n
8: end loop
```

If the simulated RAM does not halt, neither will the simulating RAM_0, but if it halts after n steps, the RAM_0 will terminate after having completed $O(n)$ simulation steps and $O(\log n)$ iterations of the loop, for a total of $O(n)$. □

3 Real time simulation

The proof of Theorem 3 is more involved, because it is no longer possible to have an $O(n)$ preparatory step. For this proof, we require the vector

$$O^n_{a,m} \stackrel{\text{def}}{=} ((a \leftarrow nm) \dot{-} a)/((1 \leftarrow m) \dot{-} 1).$$

We note that this vector is equivalent to $a \times O^n_{1,m}$, but makes no use of multiplication, except for the multiplication "nm" which can be eliminated by the same technique as was used in the proof for Lemma 1, if m is a known power of 2. For $a < 2^m$, the resulting vector is $[a, \ldots, a]$.

Also, we require the following lemma.

Lemma 2 *It is possible to convert, in $O(1)$ operations from the set $\{\dot{-}, \div, \times, \leftarrow, \wedge\}$, a vector (m, V, n) to a vector (m', V', n) for any choice of m' so as to retain the original vector elements (given that m' is large enough to store the values in V). Furthermore, if m' is a large enough known power of 2, this can be done with the operation set $\{\dot{-}, /, \leftarrow, \wedge\}$ only.*

Proof We handle three cases:

Case 1: $\boldsymbol{m' \geq m(n+1)}$ This is done by $V' = O^n_{V,m'\dot{-}m} \wedge O^n_{(1\leftarrow m)\dot{-}1,m'}$.

Case 2: $\boldsymbol{m'n < m}$ This is done by $V' = V \bmod ((1 \leftarrow m) \dot{-} (1 \leftarrow m'))$, where $a \bmod b$ is calculated as $a - (a \div b) \times b$. To see this, recall that V is the value of the polynomial whose coefficients are the vector elements $k_0, \ldots, k_{n-1}$ at the point $Z = 2^m = 1 \leftarrow m$. In order to evaluate the polynomial at $Z' = 2^{m'}$, we take its value modulo $Z - Z'$. Because $Z \equiv Z' \pmod{Z - Z'}$, for any polynomial p, $p(Z) \equiv p(Z') \pmod{Z - Z'}$. The condition of Case 2 ensures that the desired V' is smaller than $Z - Z'$, so this residue can be evaluated without ambiguity.

Case 3: **none of the above** To handle this, first calculate (m'', V'', n) for a sufficiently large m'', using Case 1, then lower to the final m' using Case 2.

If m' is large enough, only the first case is needed, so no integer division is required (only exact division). Furthermore, if m' is known to be 2^k then the multiplication nm' in the

calculation of $O^n_{(1\leftarrow m)\dot{-}1,m'}$ can be replaced by $n \leftarrow k$. For the calculation of $O^n_{V,m'\dot{-}m}$, n can be replaced by any value that is at least as large as n, without affecting the result after the $\wedge$ operation. As a result, the multiplication $n(m'\dot{-}m)$ can be replaced by $(m'\dot{-}m) \leftarrow n$, effectively replacing n by 2^n. □

We now continue to prove the main theorem.

Proof (of Theorem 3) To prove the theorem, one only needs to prove that the operations "LOAD" and "STORE" can each be simulated in O(1), after a one-time initialization. (The initialization receives no parameters, and therefore necessarily works in constant time.) This statement was made rigorously in Lemma 2 of [2]. In essence, the RAM_0 is able to compute exactly as the RAM does. Its only difficulty is in reading the inputs to the computations from the registers and writing the computation results back to the registers. Therefore, if it can simulate loading and storing properly, the rest of the problem is solved trivially.

Our simulation of the new "LOAD" and "STORE" will work very much like the "LOAD" and "STORE" of Lemma 1. However, because we can no longer anticipate the size of integers the program will be required to handle (The RAM's first command may be a "READ" instruction, which will read from the user an integer of arbitrary size), we cannot use $m = M$ directly, as was done before. Instead, we begin by using $m = 1$, which is sufficient to store the initial all-zeroes state of the registers, then increase m whenever a "STORE" command requests storage of a value greater than or equal to 2^m.

In case a value $r, r \geq 2^m$, needs to be stored, the RAM_0 changes the width of the encoded vector, as per Lemma 2, from m to $m' = \max(m(n+1), r)$. The new width is necessarily large enough to ensure Lemma 2 can be implemented with a restricted set of operations. If the new width is not a known power of 2, we can use $1 \leftarrow m'$ as a width, instead of m'.

In this way, in O(1) operations we are able to widen the vector enough to allow the simulation of any assignment.

Correspondingly, we also update n to signify the greatest register address used thus-far in the program's execution. Updating n is a simple "max" operation, affecting neither m nor V. □

The proof of Theorem 2 is applicable not only to $op = \{+, \dot{-}, \leftarrow, \rightarrow\}$, but also to any $op \supseteq \{+, \dot{-}, \leftarrow, \rightarrow\}$, given the restriction that there exists a $\text{RAM}_0[op]$ that calculates any number at least as large as $M_{op}(n, input)$ in $O(n)$ steps. This condition holds for any reasonable set of operations. For the proof of Theorem 3, any $op \supseteq \{+, \dot{-}, /, \leftarrow, \rightarrow, \wedge\}$ will do, with no added restriction.

We have shown, therefore, that for RAMs with a sufficient set of operations, the use of indirect addressing entails no advantage.

Acnowledgments The author would like to thank an anonymous reviewer for helpful comments.

References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addision-Wesley Publishing Co., Reading (1975)
2. Ben-Amram, A.M., Galil, Z.: On pointers versus addresses. J. Assoc. Comput. Mach. **39**(3), 617–648 (1992)
3. Ben-Amram, A.M., Galil, Z.: On the power of the shift instruction. Inf. Comput. **117**, 19–36 (1995)
4. Mansour, Y., Schieber, B., Tiwari, P.: A lower bound for integer greatest common divisor computations. J. Assoc. Comput. Mach. **38**(2), 453–471 (1991)

5. Mansour, Y., Schieber, B., Tiwari, P.: Lower bounds for computations with the floor operation. SIAM J. Comput. **20**(2), 315–327 (1991)
6. Paul, W., Simon, J.: Decision trees and random access machines. In: Logic and Algorithmic (Zurich, 1980), Monograph. Enseign. Math., vol.30, pp. 331–340. Univ. Genève, Geneva (1982)
7. Schönhage, A.: On the power of random access machines. In: Automata, Languages and Programming (Sixth Colloq., Graz, 1979), Lecture Notes in Comput. Sci., vol.71, pp. 520–529. Springer, Berlin, (1979)
8. Simon, J.: On feasible numbers (preliminary version). In: Conference Record of the Ninth Annual ACM Symposium on Theory of Computing (Boulder, Colo., 1977), pp. 195–207. Assoc. Comput. Mach., New York, (1977)
9. Simon, J.: Division in idealized unit cost RAMs. J. Comput. Syst. Sci. **22**(3), 421–441 (1981). (Special issue dedicated to Michael Machtey)
10. Turing, Alan M.: On computable numbers, with an application to the Entscheidungsproblem. Proc. Lond. Math. Soc. **42**, 230–265 (1936)