Unit-Cost Pointers versus Logarithmic-Cost Addresses

Amir M. Ben-Amram

Computer Science Department, Tel-Aviv University, Tel Aviv 69978, Israel

ABSTRACT

The LISP Machine (LM) is a high-level model of computation using a linked memory structure. The Hierarchical Memory Model (HMM) has a random access memory but takes into account the cost of memory access. We show that the HMM can be simulated by the LM in real time. On the other hand, for simulating an on-line LM program of time t and space s by the HMM we prove time bounds of $O(t \log s)$. These are shown to be tight for data types which are incompressible — an information-theoretic notion, allowing for models which handle a variety of data types.

0. Introduction

What should be the capabilities of an abstract "computer" for theoretic study? This question is of fundamental importance in complexity theory. The quest for bounds on the complexity of problems calls for a machine model which is both realistic and theoretically accessible. The random access machine [2,6,10] seems to be the machine model in widest use. The selection of the data type that the machine should use—its data items and the primitive functions that may be applied to them—is an important decision, and since the first use of the RAM for complexity analysis, authors have considered this question, attempting to justify or evaluate possible choices [e.g. 6,8,11].

None of these authors paid special attention to the basic capability of indirect addressing. However, including it in our instruction set is a very important decision, since the use of indirect addressing in problems where the data can be used—directly or indirectly—for addressing, makes the RAM very powerful (and hard to analyze). It allows for striking algorithms, such as distribution sort and hashing techniques. Some of these results may be attributed to the assignment of uniform cost to memory access, being unrealistic. It has been claimed that assigning another complexity measure (usually logarithmic) to memory addressing makes the RAM model closer to physical reality. This modification affects the running time of algorithms and was proposed by Aggarwal et al. [1] under the name of Hierarchical Memory Model (HMM). Under this model, all instructions cost one except for memory access instructions, which are charged according to the address accessed.

The study of random access machines and the HMM falls naturally into the framework of *high level models* [3]. Such a model is composed of a processing unit, which operates on a data type denoted by $\mathcal{T}$, and a memory unit, which varies in structure and associated costs. The RAM and the HMM differ only in these costs. In [3], Ben-Amram and Galil study a third high level model, *the LISP machine* (LM), whose memory structure supports no form of indirect addressing; it is a "pointer machine" [12,13]. The $\mathcal{T}$-LM is compared to the $\mathcal{T}$-RAM by evaluating the time complexity of on-line simulation of each of these models by the other. It is easy to simulate the LM on the RAM in real time; for LM simulation of a RAM program of time t and space s, an upper bound of $O(t \log s)$ is shown, and a matching lower bound is proved for all data types that are *incompressible*. The incompressibility property of a data type holds whenever an n-tuple of data cannot be encoded in an $n-1$-tuple, and decoded from it, within a time bound that depends only on n. Incompressibility was demonstrated in [3] for the data types of finite words, unbounded integers and real numbers, all of them with generous instruction sets. For unbounded integers, the instruction set excludes *right shift* (but includes left shift or exponentiation, multiplication and bitwise operations). It has been shown that with a right shift instruction, compression is possible and, moreover, the RAM can be simulated by a machine with finitely many registers in $O(\alpha(s))$ time per operation, where α is a (very slow growing) functional inverse of Ackermann's function [4].

In this note, we complement [3] by studying the relations between HMMs and LISP machines. The LM is shown to be stronger than the HMM; it can simulate the HMM in real time. On the other hand, assuming incompressibility of the data type, we can obtain an $\Omega(t \log s)$ lower bound on the time required by the HMM to simulate the LM. Matching upper bounds are given by simulation algorithms. The first algorithm requires $O(\log s)$ time per operation, where s is the number of LISP cells created by the LM program. Since the size of the memory structure actually used by the program may remain much lower, due to deletion of nodes, this measure may seem for certain programs exaggerated and we give a second algorithm running in $O(\sum_{t=1}^{T} \log s_t)$ total time, where s_t is the *instantaneous space* used by the LM at time $1 \leq t \leq T$, or $O(\log s_{\max})$ time per operation where $s_{\max}$ is defined as the maximum instantaneous space attained by the program. The saving in time is obtained by performing *storage compactions* at appropriate intervals, since the HMM benefits from shrinking the range of addresses accessed. For the compressible data type of integers with shift, the simulation can be made much faster; actually, we can substitute α for log in our upper bounds.

1. Definition of the Models

We begin by defining a *data type* $\mathcal{T}$ to be a triple $\mathcal{T} = (\mathcal{D}, \mathcal{F}, <)$ where $\mathcal{D}$ is the *domain* of data values, $\mathcal{F}$ the set of *primitive functions* which can be computed in a single instruction, and $<$ an order relation on $\mathcal{D}$. The operations of a $\mathcal{T}$-machine include comparison (using the given order relation), and the "arithmetic" instructions given by $\mathcal{F}$. We assume that $\mathcal{D}$ includes the integers, on which $<$ is the natural order and that the operations of addition and subtraction are provided in $\mathcal{F}$.

The storage of a LISP machine consists of data values (called *atoms*), and *dotted pairs* which are nodes of out-degree two, with outgoing arcs (CAR and CDR) that point either to atoms or to other dotted pairs. The basic instruction set includes instructions for node creation (CONS) and retrieval of the CAR and CDR of a node.

Additional instructions, allowing for redirection of the CAR or CDR pointers of an existing node, may also be included. LISP speakers will recognize that the inclusion of these instructions changes the language from **pure LISP** to standard (full) LISP. We call the pure LISP machine a PLM and the full LISP machine an FLM. The PLM is a restricted form of the FLM and thus may only be weaker than it; the relationship between the two constitutes an open problem.

A $\mathcal{T}$-RAM has for storage a linear array of "cells", each containing a single data value. The cells are addressed by positive integers. The $\mathcal{T}$-HMM is a variant of the $\mathcal{T}$-RAM, where each *memory access instruction* costs $\lceil \log a \rceil$ time units where a is the address accessed. That logarithmic address cost is "realistic" is supported by the fact that actual VLSI memories have logarithmic access delays [9, p.321], in contrast with their ability to send and receive full data words at one time. The logarithmic address cost also seems to model the hierarchical structure of memory in modern computer systems, in which there is often a small amount of fast memory augmented with increasingly larger amounts of slower memory. The logarithmic address cost can be viewed as partitioning the memory into levels, where level i contains 2^i locations and takes i time units to access. Neglecting the number of registers can be justified by the fact that a fixed number of $O(1)$ access time cells can be added to a program without increasing appreciably the access time for the other locations. Another feature which demonstrates the versatility of the model, and will be useful in this note, is our ability to program in tracks: suppose we are given k programs for the HMM. We can partition the memory into k *tracks* by assigning the addresses a congruent to i modulo k to the ith program. Whenever this program originally refers to memory location x, it will now use $xk + i$. This way these programs can be run concurrently at only a constant time overhead per memory access.

Recall that for the RAM, space complexity is defined (in the unit-cost measure) as the number of distinct addresses used by the program. The same measure can be used for the HMM. The corresponding quantity for the LM would be the number of dotted cells created, or the number of CONS instructions performed (which is the same), throughout the program [7,9]. We call this number *the accumulated space complexity* of the LM program. We later discuss different measures of space.

Let P be a program for some computational model. A program P' for a possibly different model is said to simulate P in *real time* if it simulates each step of P which takes time t within $O(t)$ time.

In Section 2 we show that the PLM can simulate the HMM in real time. Section 3 deals with the reverse problem, and it is shown that the HMM needs $\Omega(t \log s)$ time to simulate a PLM program of time t, assuming incompressibility of the data type. In Section 4 we discuss algorithms for simulating the LM on the HMM. The running time of the simulation is $O(t \log s)$, where s is a measure of space complexity for the LM.

2. Simulating the HMM by a PLM

Theorem 1. *The $\mathcal{T}$-PLM can simulate the $\mathcal{T}$-HMM in real time.*

In order to simulate the HMM in real time, it is sufficient to show a PLM data structure that simulates a random access storage unit in time of $O(\log a)$ per operation, where a is the address accessed. We use a data structure consisting of a sequence of balanced trees $T_1, T_2, \ldots$ where T_i represents the ith memory level (in the sense of the previous section). Each tree can be used to store the contents of these locations and retrieve the contents given the address; any of the standard balanced search tree algorithms will do (implementation considerations for a PLM are given in [3]). The trees are strung by their roots to a list as shown in Figure 2.1. The list is initially empty; when an operation attempts to access a tree that does not exist yet, its root is created. Afterwards, nodes are inserted into the tree when called for by a memory reference.

It is not hard to see how each operation takes $O(\log a)$ time to follow the linked

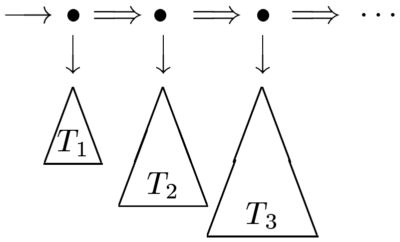

Figure 2.1

list of trees and additional $O(\log a)$ time units in handling the tree.

3. HMM simulation of the LM — a Lower Bound

Theorem 2. *If $\mathcal{T}$ is an incompressible data type, then a $\mathcal{T}$-HMM requires $\Omega(t \log s)$ time in the worst case to simulate a $\mathcal{T}$-PLM program of time t and accumulated space s.*

Proof. We study the complexity of programs for the following *list reversal* problem:

Input: $x_1, x_2, \ldots, x_n \in \mathcal{D}$.

Output: $x_n, x_{n-1}, \ldots, x_1$.

This problem can be solved on the PLM with a very simple program using space n and time cn for some constant c. We prove that any HMM program for list reversal requires $\Omega(n \log n)$ time. This proves the theorem in the case $t = cs$, and in fact for $t = O(s)$ in general.

Let s be the space complexity of a list reversal program for the HMM. The memory contents of the list reversal program after reading in all the data can be seen as an encoding of the input, since it allows its reconstruction. Thus we have a program that encodes the input n-tuple in an m-tuple of values where $m = O(s)$. A decoding program is immediate. Thus incompressibility ensures that $m \geq n$, hence $s = \Omega(n)$. Finally, it is not hard to show that any HMM program whose space complexity is $\Omega(n)$ has running time $\Omega(n \log n)$.

To complete the proof of the theorem, let $t > cs$ be arbitrary; we consider a PLM program which reverses t/cs lists of s elements each, one after another. Simulating this program by the HMM clearly requires $\Omega(t \log s)$ time. $\square$

4. Simulation of the LM by the HMM

In this section we show upper bounds for simulation of the LM by the HMM. The running time of the simulation will be $O(t \log s)$ where s relates to the "space complexity" of the LM program.

Theorem 3. *An FLM program of running time t and accumulated space s can be simulated by the HMM in $O(\log s)$ time per operation.*

Proof. This upper bound is achieved by a straight-forward simulation that assigns new space in memory for every dotted pair created. $\square$

A characteristic property of *pointer machines* is that nodes created by the program can become unreachable since all pointers to them have disappeared. Such nodes play no subsequent role in the computation, and therefore can be regarded as ceasing to exist [3,7,12]. It may seem exaggerated to consider all the nodes created in the "space complexity" of the program, when actually it may be dealing with much smaller memory graphs. We define the *instantaneous space* s_t as the number of nodes in existence at time t, including the registers. A plausible definition for the *space complexity* of the program as a whole is the maximum value of s_t, which we call $s_{\max}$.

In order to evaluate the running time of the HMM simulation with respect to the instantaneous space measure of the LM program, we define for the HMM its own instantaneous space, s'_t. This will be the "high water mark" of the addresses used in the simulation; that is, supposing that the HMM allocates some addresses to store the contents of LM memory cells and registers, s'_t is the highest address allocated. This "high water mark" can be reset by a process known as *storage compaction*, which includes identifying all the storage locations which are no longer needed (because the cells they represented disappeared), and relocating the other cells so that they occupy a contiguous block of s_t locations. The task of identifying the reusable locations is known as *garbage collection*; the compaction of the active storage is done by copying the active cells into the first s_t location of a second track (see §1). By using two tracks for the simulation, and copying the active memory alternately from one track into the other, we can apply storage compaction as many times as we want. The cost of this addition is only the time of the compaction procedure. Let T be the total running time of an FLM program. If compaction had cost nothing, we could perform a memory compaction after each FLM step we simulate and so achieve a running time of $O(\sum_{t=1}^{T} \log s_t)$. The next theorem shows that we can achieve this time bound even when taking into account the cost of compaction tasks.

Storage compaction algorithms are quite standard. One algorithm that fits our purpose can be found in Cheney [5]. This algorithm scans the memory structure in the active track and copies them to contiguous locations in the other track. Its running time is $O(s_t \log s'_t)$, since it performs a number of operations linear in s_t and the highest location it accesses (in each of the tracks) is s'_t.

Theorem 4. *The HMM can simulate an FLM program of time T in $O(\sum_{t=1}^{T} \log s_t)$ time.*

Proof. We show a simulation method that achieves the running time. The main idea is using tracks, and performing a storage compaction after (simulated) time steps $t_0 = 0$, $t_1 = t_0 + s_{t_0}$, $\ldots$, $t_{i+1} = t_i + s_{t_i}$. For simplicity we assume that $T = t_k$ for some k, that is, the simulation time can be divided into a sequence of intervals of the form $t_i + 1, \ldots, t_{i+1}$. In simulating the ith interval, the HMM performs one storage compaction and $t_{i+1} - t_i = s_{t_i}$ FLM operations. Since the storage is compacted just before the first operation, we start the interval using s_{t_i} locations, and in the next s_{t_i} operations s'_t can get no larger than $2s_{t_i}$. Therefore, simulating the FLM operations in this interval takes $O(s_{t_i} \log s_{t_i})$ time. The compaction procedure takes $O(s_{t_i} \log s'_{t_i})$ time, which for similar reasons is $O(s_{t_{i-1}} \log s_{t_{i-1}})$. Let T_{sim} denote the total simulation time. Then we have (ignoring constant factors)

$$T_{\text{sim}} = \sum_{i=0}^{k-1} s_{t_i} \log s_{t_i} + \sum_{i=1}^{k-1} s_{t_{i-1}} \log s_{t_{i-1}} = O\left(\sum_{i=0}^{k-1} s_{t_i} \log s_{t_i}\right).$$

The theorem follows by showing that

$$\sum_{i=0}^{k-1} s_{t_i} \log s_{t_i} = O\left(\sum_{t=0}^{T-1} \log s_t\right).$$

We omit the calculation. $\square$

We can further change the simulation to perform memory compactions in time-bounded stages, paired with the simulated program steps so that the overhead is uniformly distributed — $O(\log s_{t_{i-1}})$ time for every instruction in the ith interval. The straight-forward simulation of the FLM instructions is modified to make use of the newer copy of a cell which the compaction procedure has already passed.

Corollary 1. *The HMM can simulate an FLM program of maximum instantaneous space $s_{\max}$ in $O(\log s_{\max})$ time per operation.*

5. The Case of Integers with Shifts

Ben-Amram and Galil [4] give an algorithm for simulating the RAM on a machine equipped with just a finite number of registers, exploiting the power of shift instructions. The simulation time is $O(\alpha(s))$ per operation, where α is a (very slow growing) functional inverse of Ackermann's function.

Theorem 5. *Let $\mathcal{T} = (\mathbb{N}, \mathcal{F}, >)$ where $\mathcal{F}$ includes addition, subtraction, left and right shift and Boolean* AND. *Then the $\mathcal{T}$-HMM can simulate the $\mathcal{T}$-FLM within the following time bounds:*

(i) $O(t\alpha(s))$ *uniform time, where* s *is the accumulated space of the LM program.*

(ii) $O(\sum_{t=1}^{T} \alpha(s_t))$, *where* s_t *is the instantaneous space of the LM program.*

(iii) $O(t\alpha(s_{\max}))$ *(uniform), where* $s_{\max}$ *is the maximal instantaneous space throughout the program.*

Proof. The machine considered in [4] can be efficiently simulated in the HMM model, because it accesses only a constant number of registers (so each access costs a constant even on the HMM); the arithmetic instructions are the same and have unit cost in both models. Thus the HMM can use the algorithm of [4] for simulating a RAM. Using this simulation, the tth memory access will cost $O(\alpha(s'_t))$ instead of the usual logarithmic cost. Substituting this access time in the previous proofs results in the above theorem. □

6. Conclusion

The RAM, LISP machine and Hierarchical Memory Model are high-level models, defined generally with respect to data types, and apt for the design and analysis of practical algorithms. We do not attempt a definite claim regarding the choice of a computational model; each of the three we mentioned (and others) has its own merits. In the HMM, complexity analysis is somewhat harder than in the other two, but it does seem closer to physical reality of actual computers: unbounded memory is available but has its cost. LISP machines have a theoretical advantage in simplicity and clarity, and in fact many programs fall within the model since they use linked data structures. The cost of simulating this abstract model on the HMM relates to the cost of embedding pointer structures in realistic memory systems.

We have seen that the LM is stronger than the HMM (simulates it in real time). As for simulation of the LM by the HMM, we obtained an upper bound of $O(\log s)$ time per operation, where s is the *maximum instantaneous space measure*, and proved this tight for all incompressible data types.

We conclude that the HMM is asymptotically slower than the LM by a factor of $\log s$ (in plausible settings). In [3], we obtained a similar gap between the LM and the RAM. The gap may shrink with a powerful data type (such as the integers with *shift*), but this might actually testify against assuming this data type in analysis. Even in the former case there are specific problems where two of the models perform equally well, or even the three of them do [1,10], and as suggested in [1], it is instructive to classify problems on the grounds of their complexity differences between models. As for the worst-case gap between the HMM and RAM, we can deduce from the above that it lies between $O(\log s)$ and $O(\log^2 s)$ per operation; we pose the determination of tight bounds as an open problem. The main observation is that obviously, an HMM simulating a RAM must build a data structure, which should fit in low addresses and describe the contents of the RAM's memory. Standard ways to do it, by means of search trees, lead to $O(\log^2 s)$ cost [1].

Acknowledgment

Comments from Uri Zwick are gratefully acknowledged.

REFERENCES

[1] A. Aggarwal, B. Alpern, A. K. Chandra and M. Snir, "A model for hierarchical memory," in: *Proc. 19th Annual ACM Symp. on Theory of Computing* (1987), 305–314.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley, Reading, Mass., 1974).

[3] A. M. Ben-Amram and Z. Galil, "On pointers versus adresses," *J. of the ACM* 39:3 (1992) 617–649.

[4] A. M. Ben-Amram and Z. Galil, "On the power of the shift instruction," to appear in *Information and Computation* (1994).

[5] C. J. Cheney, "A non-recursive list compacting algorithm," *Comm. of the ACM* 13:11 (1970) 677–678.

[6] S. A. Cook and R. A. Reckhow, "Time bounded random access machines," *J. Comput. System Sci.* 7:4 (1973), 354–375.

[7] P. van Emde Boas, "Space measures for storage modification machines," Information Processing Letters 30 (1989) 103–110.

[8] D. Harel and R. E. Tarjan, "Fast algorithms for finding nearest common ancestors," *SIAM J. Comput.* 13 (1984), 338–355.

[9] C. Mead and L. Conway, *Introduction to VLSI Systems* (Addison-Wesley, Reading, Mass., 1980).

[10] R. Paige, "Real-time simulation of a set machine on a RAM," in: *Computing and Information ICCI '89*, Vol. 2, R. Janicki and W. Koczkodaj (eds.), 1989.

[11] F. P. Preparata and M. I. Shamos, *Computational Geometry: an Introduction* (Springer-Verlag, New-York, 1985).

[12] A. Schönhage, "Storage modification machines," *SIAM J. Comput.* 9:3 (1980), 490–508.

[13] R. E. Tarjan, "A class of algorithms which require nonlinear time to maintain disjoint sets," *J. Comput. System Sci.* 18 (1979), 110–127.