On the Power of the Shift Instruction

Amir M. Ben-Amram* Zvi Galil** Tel-Aviv University
Tel-Aviv University and Columbia University

Postponement is the thief of time.
Seven Star Diary 1989

ABSTRACT

This paper examines the power of the shift primitive when included in a high level model operating on unbounded integers. It is shown that in such a model a constant number of registers suffices for simulating an unbounded memory RAM of the same instruction set. The simulation is on-line, and its cost can be bounded by $O(t\alpha(s))$, for a RAM program of running time t and space s. By multitask programming (postponing lengthy updates) it can be made close to real-time ($O(\alpha(s))$ per operation).

0. INTRODUCTION

The random access machine (RAM) seems to be the computational model in widest use for algorithm design and analysis (Aho, Hopcroft and Ullman, 1974). The RAM is intended to model what we are used to in conventional programming, idealized in order to be better accessible for theoretical study. Thus we let it handle "items" which are of a data type such as the integers, real numbers etc. The data type of unbounded integers is the most common, but it still permits us to select in many ways the set of *primitive operations* that our model will allow in unit time (we refer to the uniform cost criterion). Since the first use of the RAM for complexity analysis, authors have attempted to justify or evaluate possible choices of the instruction set.

Such a description of the RAM includes it in the family of *high level models* (Ben-Amram and Galil, 1992), contrasted with models such as the Turing machine and the acceptor RAM. Being the standard platform for the study of algorithms, high level models deserve thorough analysis. Ben-Amram and Galil attempt to determine the power of random access to memory, as featured in the RAM model by means of *indirect addressing*, in terms of the cost of simulating it without the use of integers as addresses; the model used for comparison is the LISP machine, whose memory consists of pointer structures. An important feature of high level models is allowing the choice of a data type, and the time bounds of (Ben-Amram and Galil, 1992) are shown to be tight for every data type which is *incompressible,* incompressibility being a feature inherent in the data type. It expresses the value of memory to a machine of this data type, in indicating that memory cannot be exchanged for (bounded) time. Among the data types proved incompressible we find the type of unbounded integers with the following set of primitives: addition, subtraction, multiplication, any Boolean (bitwise) operations, integer division by two (right shift by one

* Work supported by a Charles Clore fellowship.
** Partially supported by NSF Grants CCR-90-14605 and CISE Institutional Infrastructure Grant CDA-90-24735.

bit), left shift by an arbitrary integer and even integer exponentiation (which subsumes left shift). This is a rich repertory, extending instruction sets previously studied (Cook and Reckhow, 1973; Hartmanis and Simon, 1974; Pratt and Stockmeyer, 1976) except for instructions which allow for a right shift by an arbitrary amount. Actually, we showed that with right shift we get a compressible data type. We thus expect this addition to affect substantially the properties of the model.

The power of shift operations has been demonstrated by a series of papers on model relationships (Pratt, Rabin and Stockmeyer, 1974; Pratt and Stockmeyer, 1976; Schönhage, 1979; Simon, 1977, 1981 and 1992; Bertoni, Mauri and Sabadini, 1981). These papers show that for unit-cost RAMs with a shift instruction and either *multiplication* or Boolean operations, polynomial time is equivalent to polynomial space on a Turing machine. The addition of division makes the model even stronger. The drawback of their analysis, from the point of view of this paper, is that they consider the RAM as an acceptor of a language over the binary alphabet. But high level machines, which are supposed to model practical computations, input and output arbitrary values. Our study is concerned with such machines. Similarly, since we are interested in the running time of algorithms (and not wide complexity classes), we are not satisfied with polynomial reducibility, but look for exact complexity (at least up to constant factors).

In this paper we investigate the contribution of random access memory to the power of a model which has the capability for arbitrary shift. We show that unbounded random access memory can be simulated with a constant number of registers at the price of a low, uniform increase of the running time. Let s be the space (number of cells) used by a RAM program. Then the cost of simulating it with a constant number of registers turns out to be $O(\alpha(s))$ per operation, where α is a functional inverse of Ackermann's function. So we see that space (the number of memory registers used), which is frequently a critical resource, and is provably so for incompressible data — is no longer critical under this model. We remark that this phenomenon cannot occur when using the logarithmic space measure, which counts the number of bits stored in each register: the data type of single bits is incompressible.

In the next section we give a semi-formal definition of the random access machine and its variant, the shifting machine (SM) which has just a constant number of registers. Section 2 is concerned with precisely defining the method for comparison of the models. Following Ben-Amram and Galil (1992), we reduce the general question of simulation efficiency to the complexity of RASUS (random access storage unit simulation), that is, simulating the memory access operations of the RAM. A simplification of RASUS, *retrieval by position*, is defined, which is a useful gateway to solving RASUS. This section also states our results precisely. In Section 3 we present some programming techniques which we use on the SM. These are efficient data structures and multitask programming. In Section 4 we solve *retrieval by position*. Section 5 adds multitasking to this solution and obtains the uniform delay solution. The solution is extended in Section 6 to a full-scale RASUS. Section 7 discusses space complexity and section 8 concludes this paper.

1. DEFINITION OF THE MODELS

In this section we give a detailed description of the machines whose performance we wish to compare, the random access machine and the shifting machine. Both are high level models of computation of the same (integer) data type, and the description that follows is a specialization of the general definition given in (Ben-Amram and Galil, 1992).

A random access machine (RAM) consists of a read-only input tape, a write-only output tape, a finite number of registers and a memory. Each of the input and output tape squares, as well as the memory cells and registers, contains one natural number. These numbers also serve as memory addresses. Note that we use the term *registers* in the sense of "CPU registers", which are hard-coded in instructions, and do not refer to addressable memory cells by this term.

We program the RAM using a standard instruction set consisting of the following instruction groups. In the instructions, we denote registers by R_i, R_j.

1. *I/O instructions*:	$READ$	R_i
	$WRITE$	R_i

2. *Standard register operations*:	SET	$R_i, constant$
	SET	R_i, R_j

These instructions set R_i to a constant or to the value of R_j.

3. *Flow control instructions*:	$JUMP$	$label$
	$JEQL$	$R_i, R_j, label$
	$JGTR$	$R_i, R_j, label$
	$HALT$	

4. *Memory access instructions*:	$LOAD$	R_i, R_j
	$STORE$	R_i, R_j

These instructions access the memory cell whose address is in R_j.

5. *Arithmetic instructions*:	ADD	R_i, R_j
	SUB	R_i, R_j
	etc.	

The semantics of a RAM program need not be explained in detail. It is assumed that the RAM starts up with an all-zero memory, and advances, according to the program and the input data, between instantaneous states described by the contents of the registers, the program counter and the memory. For the running time of a RAM program we employ the *uniform cost model*, that is, we charge one "time unit" for each instruction performed. Uniform cost is also applied to space complexity, defined as the number of memory locations used.

The repertory of *arithmetic instructions* has a strong impact on the power of the machine. In particular, it is shown by Ben-Amram and Galil (1992) that an instruction set including addition, subtraction, multiplication, any Boolean (bitwise) operations, integer division by two (right shift by one bit) and left shift by an arbitrary integer constitutes (over the domain of integers) an *incompressible* data type. This means that it may be impossible to reduce the space complexity of a program without raising its running time in a non-uniform manner (that is, with no bound that depends on the size of the problem alone).

In this paper we consider machines whose instruction set contains both right and left shift. We denote by $\text{shift}(x, y)$ the result of shifting x *left* by y bits, i.e. $x \cdot 2^y$. We denote by $\text{shift}(x, -y)$ the result of shifting x *right* by y bits, i.e. $\lfloor x/2^y \rfloor$.

The other instructions we assume are addition, subtraction, multiplication and bitwise AND. The last two can be elininated; but it is convenient to include them. The impact of their removal is discussed in the last section.

The *shifting machine* (SM) is a simple variant of the RAM. It differs from the latter by having no "storage unit": only the finite set of registers is available for holding information. Using the results of the following sections, it is possible to determine a constant number of registers which suffices for simulating any larger, constant number, in real time. This justifies assuming any convenient number of registers for an SM program.

2. THE SIMULATION PROBLEM

2.1 Simulation of the Storage Unit

As the only difference between the RAM and the SM is the lack of random access memory in the latter, it is natural to question whether the SM can simulate the use of memory by the RAM; and if so, at what cost. As we shall see, the simulation is possible; and the cost of solving this problem translates to a general bound on the cost of simulating an on-line RAM program by our machine. This problem is the *random access storage unit simulation* discussed in (Ben-Amram and Galil, 1992) and we repeat the definitions and results thereof.

A *simulation of the RAM storage unit* (RASUS for short) consists of two programs for a high level model. One program is used to carry out a STORE and the other to carry out a LOAD. Initially the machine starts with a fixed memory structure that corresponds to an all-zero memory of the RAM. Two registers are designated as a and d, where the *address* and *datum* are placed as input to the *store* procedure; while for *load*, an address is placed in a, and the task of the procedure is to deposit in d the value last "stored" into address a. The contents of memory cells and registers remain unchanged between consecutive invocations of the programs. Instead of providing the initial memory structure, it may be convenient to have an *initialization program* in addition to the previous two. This is a one-shot program with no input, so its running time is fixed and of no interest.

Let $T_{sim}(u)$ be the total time spent by the machine in performing the simulation for the sequence of commands u. Define the <u>RAM time</u> $t(u)$ to be the number of instructions in u, i.e. its length; and the RAM space $s(u)$ to be the number of distinct addresses that appear in STORE commands in u. Then we measure the *time complexity* of the simulation by the function

$$T(t,s) = \sup_{u:\, \substack{t(u)\le t \\ s(u)\le s}} T_{sim}(u).$$

We can also consider $T_{load}(t,s)$ and $T_{store}(t,s)$, the worst-case time needed by the *load* and *store* programs, respectively, after any preceding command sequence u such that $t(u) \le t$ and $s(u) \le s$.

If $T_{load}(t,s)$ and $T_{store}(t,s)$ are bounded by the function $f(t,s)$, we say the simulation has a *delay* of $f(t,s)$. We further attribute it as *uniform delay* if $f(t,s) = O(\frac{T(t,s)}{t})$. A simulation having constant uniform delay is said to be *real time*.

2.2 Machine Simulation

We define one program as simulating another if their external behaviour matches. By external behaviour we mean the sequence of input instructions performed, output produced (for a given input) and halt (if done) throughout the execution of the program. Note that this is necessarily an *on-line* simulation (the alternation of input and output operations is preserved). Let $t_0 < t_1 < \cdots < t_i < \cdots$ be the time steps at which the simulated program makes an external operation. Let $t'_0 < t'_1 < \cdots < t'_i < \cdots$ be the corresponding time steps of the simulation. Let s_t be the space used by the RAM in the time interval $[0,t]$. Then we say that the simulation runs in time $T(t,s)$ if for all input sequences, for all i such that $t_i \le t$ and $s_{t_i} \le s$, we have

$$t'_i \le T(t,s).$$

The simulation is said to be *real time* when there exists a constant $c > 0$ such that for all input sequences we have

$$t'_i - t'_{i-1} \le c(t_i - t_{i-1}) \qquad \forall i \ge 0.$$

In general we say that the simulation has *uniform delay* if there is a function $f(t,s) = O(\frac{T(t,s)}{t})$ such that for all input sequences,

$$t'_i - t'_{i-1} \le f(t_i, s_{t_i}) \cdot (t_i - t_{i-1}) \qquad \forall i \ge 0.$$

2.3 Equivalence of the Problems

The following is proved in (Ben-Amram and Galil, 1992):

The Simulation Lemma. *General simulation of the RAM and the RASUS problem are real-time equivalent on any high level model. That is, if a machine can solve RASUS with time complexity $T(t,s)$, then it can simulate every RAM program in time governed by $3T(t,s)$. Conversely, if there exists a function $T(t,s)$ such that every RAM program can be simulated by our model in time governed by $T(t,s)$, then it can solve RASUS in time complexity $T_{RASUS}(t,s) \le cT(t,s)$ for some constant c. With respect to unifrom delay solutions, the same statement applies to the delay.*

2.4 Retrieval by Position

Retrieval by position is a simplification of the RASUS problem, in which LOADs are random, but STOREs are sequential. This problem contains (in our case) the hard core of RASUS. Our RASUS algorithm will be based upon an algorithm for retrieval by position.

A solution to the latter problem consists, as for RASUS, of a pair of procedures, *store* and *load*. Contrary to RASUS, *store* does not accept an address; rather, data "stored" are identified by their position in the input sequence. Let $a_0, a_1, \ldots, a_{n-1}$ be the numbers read by *store* during a run. The *load* procedure accepts an integer $0 \le i < n$ and should respond with an output of a_i. The time complexity of the solution is determined as for a RASUS.

2.5 Statement of Results

For $i \ge 1$ and $j \ge 0$ the Ackermann function $A(i, j)$ is defined by

$$\begin{aligned} A(i,0) &= 2 && \text{for } i > 1; \\ A(1,j) &= 2^j && \text{for } j \ge 0; \\ A(i,j) &= A(i-1, A(i, j-1)) && \text{for } i > 1,\ j \ge 1. \end{aligned}$$

Let

$$\alpha_k(s) = \min\{\, j \mid A(k,j) > s \,\},$$
$$\alpha(s) = \min\{\, k \mid A(k,k) > s \,\}.$$

Actually, Ackermann's function appears in the literature in several variants, yet the diffrences are not essential (the above definitions are most similar to those in Hart and Sharir, 1986 and Tarjan, 1983). The functions α_k increase more slowly as k increases, beginning with $\alpha_1(s) = \lfloor \log s \rfloor + 1$ and $\alpha_2(s) = \lfloor \log^* s \rfloor$. In fact, Ackermann's function has the property that every primitive recursive function is majorized by $A(k, j)$ for some k (Péter, 1967). Therefore, for every primitive recursive function f some α_k grows slower than f^{-1}. $A(k, k)$ grows faster than any primitive recursive function, and its inverse α grows slower than every inverse of such function. α grows so slowly that $\alpha(s) \le 3$ for all $s < A(3, 3)$, a huge number.

A well-known measure for the performance of an on-line algorithm is *amortized complexity*. It is defined as the cost of a worst-case sequence of operations, divided by the length of the sequence. The following theorem characterizes the best amortized running time that we obtain for RASUS.

Theorem 1. *The SM can solve RASUS with $T_{store} = O(1)$ and $T_{load} = O(\alpha(s))$, or $T_{load} = O(k)$ and $T_{store} = O(1 + \log \alpha_k(s))$, for any fixed k, where in both cases, the store time is amortized.*

With respect to worst-case complexity, we obtain a close result.

Theorem 2. *The SM can solve RASUS in $O(\alpha(s))$ time per operation. Hence, the SM can simulate the RAM in $O(t\alpha(s))$ uniform time.*

3. PROGRAMMING TECHNIQUES

3.1 Preliminaries

In this section we present basic programming techniques for the SM, and define data structures that are fundamental to the RASUS algorithm.

Let the binary representation of x be $b_n \ldots b_1 b_0$. We denote by $\mathrm{Right}_d(x)$ the right d bits of x, i.e. the number $b_{d-1} \ldots b_1 b_0$. Right_d is computable by

$$\mathrm{Right}_d(x) = x - \mathrm{shift}(\mathrm{shift}(x, -d), d).$$

By $\mathrm{Mid}_{i,d}(x)$ we denote the contents of the d-bit block starting at bit i of x:

$$\mathrm{Mid}_{i,d}(x) = \mathrm{Right}_d(\mathrm{shift}(x, -i)).$$

We can modify the contents of this block to a given value, without affecting the rest of the number x, in constant time:

Lemma 1. *Let $0 \leq x < 2^b$. Let $y \in \mathbb{N}$ such that $\mathrm{Mid}_{i,b}(y) = x_1$. And let $z = y - \mathrm{shift}(x_1, i) + \mathrm{shift}(x, i)$. Then $\mathrm{Mid}_{i,b}(z) = x$ and for all $j \notin \{i, \ldots, i+b-1\}$, the jth bits of z and y are the same.*

If we know that $\mathrm{Mid}_{i,b}(y) = 0$, the operation is simply $z = y + \mathrm{shift}(x, i)$.

3.2 The Packet

Consider a situation in which the n data are input first; LOADs follow this stage and are not intermixed with STOREs. This problem can be solved efficiently by arranging the stored data in a *packet*.

A packet P represents an n-tuple of integers $P[0], \ldots, P[n-1]$. We write $P = P^{(n)}$ for emphasis. The packet is actually represented by two numbers, a (the *aggregate*) and b (the *block size*). We require that for all $0 \leq i < n$, $P[i] \in \{0, \ldots, 2^b - 1\}$. This enables us to pack them all into the aggregate a by letting $a = \sum_{i=0}^{n-1} \mathrm{shift}(P[i], ib)$. The packet members can be retrieved in constant time, using the relation:

$$P[i] = \mathrm{Mid}_{ib,b}(a).$$

For this operation, the value of n is not required; some of the forthcoming packet operations do need it. Yet, n will not be actually stored in the packet data structure, and will have to be given separately. This simplifies the handling of packets somewhat and incurs no difficulty in our algorithms.

Two additional operations on packets can be performed in constant time. Let $P = (a, b)$. To *update* the value of $P[k]$ to x we let

$$a \leftarrow a - \mathrm{Mid}_{kb,b}(a) + \mathrm{shift}(x, kb).$$

We require that $x < 2^b$, so that the block size remains valid. In *combining* a packet $P^{(n)} = (a, b)$ with another packet of the same block size $Q = (c, b)$ we create a new packet

$$R = \text{combine}(P, Q)$$

whose members are the members of P followed by the members of Q. This is accomplished in constant time by:

$$\begin{aligned} u &\leftarrow \text{shift}(c, nb) + a \\ R &= (u, b) \end{aligned}$$

For an *empty* packet we use $(0, b)$ for any $b \geq 0$.

Trying to make use of the packet data structure in on-line algorithms we are faced with the problem of selecting the block size. We cannot do this unless we can bound the size of all the numbers to be stored in our packet. In RASUS, at any step of the algorithm we can build a packet of numbers already stored, but it may fail to accommodate the next input number. Therefore we introduce a procedure of *scaling* the packet. Scaling means changing the block size b to a new value $d > b$. We now provide a procedure, scale(P, d), which achieves this for a packet of n members in time $O(\log n)$.

3.3 Implementation of scale(P, d).

Input: a packet $P = (a, b)$ of n members, and a new block size $d \geq b$.

Output: a packet $Q = (c, d)$ accommodating the same members, with d as block size.

Method: We make use of the parallelism in inherent in the instruction set for operating on several members of the packet at once. The basic operation of this kind is moving a set of bits inside a register, selected by a mask m. We assume that the destination of the move is vacant (contains zeros) except for possible overlap with the current location of these bits. We use the following procedure where k is the distance to move:

$$\begin{aligned} &\text{move}(a, m, k)\textbf{:} \\ &\quad t \leftarrow a \wedge m \\ &\quad a \leftarrow a - t \\ &\quad a \leftarrow a + \text{shift}(t, k) \end{aligned}$$

Clearly, *move* requires constant time.

We know that bit 0 of $P[i]$ coincides with bit ib of a. To scale P to block size d, we have to move $P[i]$ to position id. We have n members to move, which is done in $\lceil \log n \rceil$ steps. At step j we move the members $P[i]$ for which the jth most significant bit of i is a one: thus on the first step, we move the left half of the packet (Figure 3.1). Since this half is moved leftward, there is no collision with other data. This process recurs for each of the halves.

Refer to the procedure shown in Figure 3.1: we call each iteration through the loop a *scale step*. Clearly, one scale step takes constant time, and so do the instructions outside the loop.

procedure *scale*

1: $l \leftarrow \lceil \log n \rceil$
2: $m \leftarrow \text{shift}(\text{shift}(1, 2^{l-1}b) - 1, 2^{l-1}b)$
3: **while** $l > 0$
4: $\quad k \leftarrow 2^{l-1}(d - b)$
5: $\quad \text{move}(a, m, k)$
6: $\quad m_1 \leftarrow m$
7: $\quad m \leftarrow m \wedge \text{shift}(m, 2^{l-2}b)$
8: $\quad m \leftarrow m + \text{shift}(m, -2^{l-1}b)$
9: $\quad \text{move}(m, m_1, k)$
10: $\quad l \leftarrow l - 1$
11: **end while**

example: $n = 4$

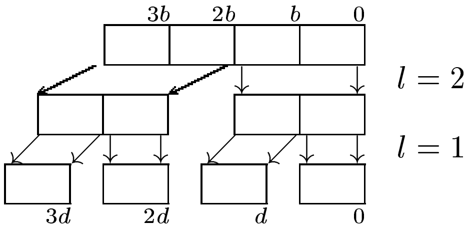

Figure 3.1

Claim 1. *Let $P = (a, b)$ be a packet of n members, and let $d \geq b$. Execution of* scale(P, d) *includes $\lceil \log n \rceil$ scale steps, hence takes $\Theta(\log n)$ time, and creates a packet (a, d) containing $P[0], \ldots, P[n-1]$.*

Proof. The procedure uses *move* both for moving the members in the packet and for adjusting the mask for the next step. Each line of the procedure (except for the first) clearly requires constant time. The computation in line 1 will be performed in constant time using a table, but for the completeness of this proof we note that $\lceil \log n \rceil$ time suffice for it anyway. The loop iterates $\lceil \log n \rceil$ times, hence the total time is $\Theta(\log n)$. A complete correctness proof is given in Appendix A. □

Theorem 3. *The packet data structure can be realized as two integers and supports the following operations:*

(i) creation of an empty packet of any block size in constant time.

(ii) retrieval of a member in constant time.

(iii) updating a member in constant time, given the block size remains valid.

(iv) combining two packets of the same block size in constant time.

(v) scaling an n-members packet to a new block size in $O(\log n)$ time.

3.4 The Map, the Stack and the Table

The basis for all SM data structures, as for the packet, is storing integer items in contiguous substrings ("blocks") of one binary integer a. A *bit-pointer* to the item x in this data structure is a pair (i, d) such that x is stored in $\text{Mid}_{i,d}(a)$. A pointer to an item allows it to be replaced in constant time by a new value, provided that the latter obeys the block size d. The *packet* achieves the functionality of a conventional *vector* by allowing a pointer to each of its members to be derived in constant time from its position. A variant of the packet, called a *map*, is used for representing a mapping $M : \{0, 1, 2, \ldots\} \to \{0, \ldots, 2^b - 1\}$, where only finitely many of the $M[i]$ are nonzero. It has the same structure as a packet:

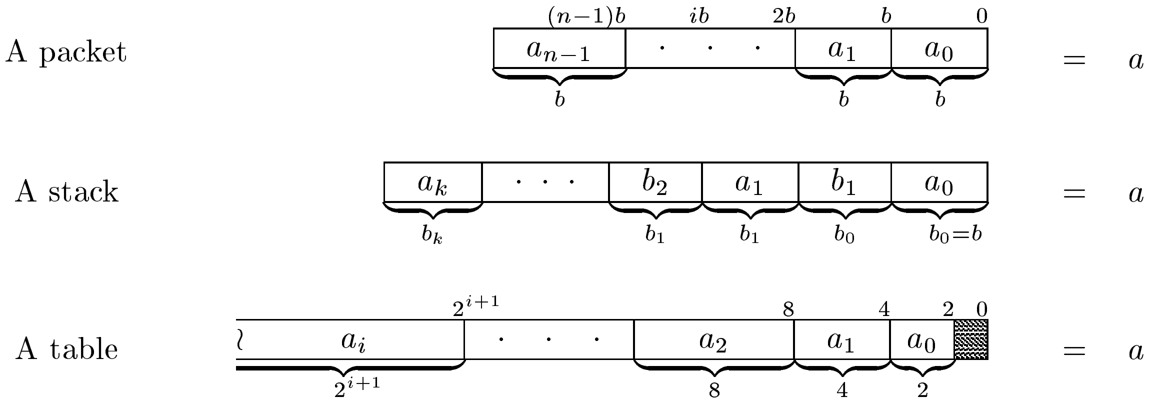

Figure 3.2

$a = \sum_{i=0}^{\infty} \text{shift}(M[i], ib)$. Hence updating and retrieval procedures are the same. But, since the number of members is not bounded, we do not apply *scale* and *combine* to maps.

A *stack* S may contain any number of members of different sizes, but only the top one is immediately accessible via the operation pop(S), which returns the top member of S and removes it from the stack. Elements are added to the stack by **push**. The stack is realized, as the packet, by two numbers (a, b) where a includes all the data and b is a block size. In contrast with the packet, b is here a block size for the top member only. *Pop* returns this element and replaces b by the block size of its successor. Our implementation of a stack appears in Figure 3.2, where a_k is the bottom member, a_0 the top of stack, and we require that $a_i < 2^{b_i}$ and $b_{i+1} < 2^{b_i}$. The programming details can be completed by the interested reader. An empty stack is represented by $(0, 0)$.

Theorem 4. *The stack data structure can be realized by two integers. It supports the following operations in constant time:*

(i) creation of, and testing for, an empty stack.

(ii) push and pop.

(iii) retrieving and updating an arbitrary member within the stack given a pointer to it.

It may be noted that our "stack" is analogous to a conventional implementation of a stack as a linked list, with nodes containing one one data item and one forward pointer. Our "pointers" are a bit weaker than conventional pointers, in that whenever members are pushed onto or popped off the stack, the positions of the members within it change. Hence pointers into the stack are invalidated.

The *table* T is another implementation of a vector, differing from the packet and map by allowing infinitely many values, with unbounded size, to be put in without restructuring. This is achieved by having the size of each member to be bounded by a (different) bound which is fixed in advance. Thus the whole table can be encoded in a single integer T, each member put into a fixed block of bits. In our applications, a bound for the size of $T[n]$

$(n \geq 0)$ will be 2^{n+1}. Hence we can use

$$T[n] = \text{Mid}_{2^{n+1},2^{n+1}}(T)$$

(Figure 3.2).

3.5 The Packet Tree

To build the desired structure we need an important observation: the data structures that we presented, the packet and the stack, can be easily adapted to handle as members, not single numbers, but k-tuples (for some constant k). The running time of all the operations on these structures will only grow by a factor of $O(k)$. In particular we can store, in any of the structures, pairs (a, b) representing packets. If P is a packets packet, we will refer to each of the member packets as to a regular element, with the notation $P[i]$.

In this way, composite data structures may be produced of simpler structures. Our fundamental composite structure is the *packet tree*. This is a multilevel structure. On the first level, we have the "data packets". These packets are packed inside the packets of the second level, and so forth, up to a level where there is a single packet. This structure corresponds in a natural way to an ordered tree. We denote the packets of level i, $i \geq 1$, by $P(i, j)$ $(j = 0, 1, 2 \ldots)$. Figuratively we describe the tree as ordered from right to left ($P(i, 0)$ is the rightmost node of level i). The packet tree allows updating of a member in constant time under due restrictions:

Observation. *Each part of a packet tree (either a simple datum or a packet) can be replaced by a new value in constant time, given a pointer to the member, provided the new value obeys the current block size of the enclosing packet.*

3.6 Multitasking

The structure of the *packet* forces us to spend a lot of time if we want to change its block size. In our algorithms we will have this need and therefore, to achieve near real-time performance, we will postpone the completion of lengthy tasks in favor of more urgent ones. We now describe the multitasking technique which serves us to this end.

A task is any sequence of program operations, which may operate on global data as well as maintain some local variables of its own. We *suspend* a task by pushing a packet of the local information that it needs for proceeding on a stack. This packet is called a *state record* (SR), and the stack is called the SR stack. All tasks will be broken into *steps* of bounded duration, such that it will be convenient to suspend a task between consecutive steps. *A scheduler round* consists of executing a bounded number of operations on behalf of the tasks in the stack. Each round will comprise up to c steps, where c is a constant to be specified later. The tasks are executed in LIFO order; that is, the scheduler pops a task SR off the stack and executes it until c steps have been performed or the task is done. In the latter case it pops another task for execution; in the former, the current values are packed in an SR which is pushed back to the stack.

As a useful example we consider a task running the procedure *scale*. We can suspend it before any *scale step* by saving in the state record the quintiple $\lambda = (l, m, b, d, a)$.

1: $B \leftarrow \max(B, a_n)$
2: $P \leftarrow (a_n, B)$
3: **for** $j = 0, \ldots, \tilde{\jmath}(n) - 1$
4: $\quad$ $\mathrm{scale}(P(1,j), B)$
5: $\quad$ $P(1,j) \leftarrow \mathrm{combine}(P, P(1,j))$
6: $\quad$ $P \leftarrow P(1,j)$
7: **end for**
8: $P(1, \tilde{\jmath}(n)) \leftarrow P$

Figure 4.1
storing a_n

4. Retrieval By Position — Single Tasking

In this section we describe a solution to retrieval by position. In the next section we will add multitasking to the algorithm, and obtain a uniform delay solution. The description of the algorithm has three parts: data structures, *load* and *store*.

4.1 Data Packets

In this subsection we describe the packets where the data stored reside. At this stage, we can ignore the rest of the structure, whose sole purpose is to ease access to the data packets. The packets we consider are denoted by $P(1,j)$, $j = 0, 1, \ldots$ (the number of packets grows as data are being added).

Definition. *For all $n \in \mathbb{N}$, $\tilde{\jmath}(n)$ is the position of the rightmost zero bit in the binary representation of n (bit positions are numbered right to left starting with zero).*

Note this is also the position the rightmost one-bit in $n+1$.

The size of $P(1,j)$ (number of members) is 2^j. Thus the packets $P(1,0), \ldots, P(1,t-1)$ will allow for storage of up to $2^t - 1$ data elements. After having stored n data, they will be contained in the packets $P(1,j)$ such that bit j in the binary representation of n is a one. Clearly the sizes of these packets sum up to n. The other packets will be logically empty (which means, that whatever their value is, it is ignored). Thus the process of adding one element (the $n+1$) to the structure can be visualized by considering the binary representation of a counter advancing from n to $n+1$: positions $0, 1, \ldots, \tilde{\jmath}(n) - 1$ change from 1 to 0 and position $\tilde{\jmath}(n)$ changes from 0 to 1. Similarly, we add the new element to the contents of $P(1,0), \ldots, P(1, \tilde{\jmath}(n) - 1)$ and create a single packet, whose size is $2^{\tilde{\jmath}(n)}$ and becomes $P(1, \tilde{\jmath}(n))$. It is convenient that the last packet created will always use the largest block size in this level. Therefore, we maintain a variable B, whose value is the current largest block size.

The procedure for adding a datum is shown in Figure 4.1 (actually it is incomplete, ignoring the manner of accessing $P(1,j)$ which will be elaborated in the sequel). Intermidiately in the process, we store in $P(1,j)$ a packet twice the intended size of $P(1,j)$ (resulting from the *combine* in line 5). For the algorithm as presented in this section, there is actually no benefit in storing it there — we could just set $P \leftarrow \mathrm{combine}(P, P(1,j))$. This act is necessary for the multitasking version. Anyway, storing a packet twice the stated size is fine as long as we treat it correctly later. We refer to this process as *an accumulation.* It operates on a range of packets, initially $P, P(1,0), \ldots, P(1,\tilde{\jmath}(n))$, gradually shrinking it until one packet remains. The left end of this range $(P(1,\tilde{\jmath}(n)))$ is called the *head* of the accumulation, and the right end (which moves towards the head) its *tail.* The accumulation ends when head and tail coincide.

Lemma 2. *Assume n values, $a_0, \ldots, a_{n-1}$ have been stored into our structure. Then a_i can be located as $P(1,h)[(2^h-1)\wedge i]$ where $h = \alpha_1(n-(n\wedge i))-1$.*

Proof. When storing a_m we join the packets P, $P(1,0)$, $\ldots$, $P(1,\tilde{\jmath}(m)-1)$ and put them in $P(1,\tilde{\jmath}(m))$. Therefore after storing $a_i, a_{i+1}, \ldots, a_{n-1}$, a_i must be in $P(1,h)$ for $h = \max_{m=i,\ldots,n-1} \tilde{\jmath}(m)$. The order in which packets are combined in the basic update procedure is "oldest first", meaning that the resulting packet contains the 2^h numbers in their insertion order. Also these are exactly the last 2^h numbers inserted to the structure up to that moment. Let these numbers be $a_k, \ldots, a_{k+2^h-1}$. Then it is the insertion of a_{k+2^h-1} that caused the creation of the current packet $P(1,h)$. It follows that $k+2^h$ is a multiple of 2^h, whence for $l < 2^h$ there holds $(k+l)\wedge(2^h-1) = l$. We conclude that the position of a_i among these elements is $i \wedge (2^h-1)$.

It remains to be shown that $\max_{m=i,\ldots,n-1} \tilde{\jmath}(m) = \alpha_1(n-(n\wedge i))-1$. We distinguish two cases. *Case 1*: $\tilde{\jmath}(n-1)$ is greater than its predecessors. This means that n is the first whole multiple of $2^{\tilde{\jmath}(n-1)}$ greater than i. The situation is as follows:

$$\begin{array}{rl} & \qquad\quad {\scriptstyle \tilde{\jmath}(n-1)} \quad\ {\scriptstyle 0} \\ i & xxxxxx\ 0\ xxxx \\ n-1 & xxxxxx\ 0\ 1\cdots 1 \\ n & xxxxxx\ 1\ 0\cdots 0 \end{array}$$

The bits left of $\tilde{\jmath}(n-1)$ are the same in n and i. We obtain

$$n-(n\wedge i) = 2^{\tilde{\jmath}(n-1)} \Rightarrow \alpha_1(n-(n\wedge i)) = \tilde{\jmath}(n-1)+1.$$

Case 2: the maximum among $\{\tilde{\jmath}(i), \ldots, \tilde{\jmath}(n-1)\}$ is obtained for the first time by $\tilde{\jmath}(l-1)$ for some $i < l < n$. Let $h = \tilde{\jmath}(l-1)$; then l is an odd multiple of 2^h and $n < l+2^h$. The situation is as follows:

$$\begin{array}{rl} & \qquad\qquad {\scriptstyle h} \quad\ {\scriptstyle 0} \\ l & xxxxxx\ 1\ 0\cdots 0 \\ \mathrm{n} & xxxxxx\ 1\ xxxx \end{array}$$

n has bit h (and those to its left) equal to those of l. Thus $n-(n\wedge i)$ is the same as $l-(l\wedge i)$ except for bits less significant than h. One of the bits h or higher must be a one in this number, because $i < l$. Since the value of $\alpha_1(x)$ is determined by the most significant one-bit of x, we have

$$\alpha_1(n-(n\wedge i)) = \alpha_1(l-(l\wedge i)).$$

Since Case 1 applies to l,

$$\tilde{\jmath}(l-1) = \alpha_1(l-(l\wedge i))-1 = \alpha_1(n-(n\wedge i))-1. \quad \square$$

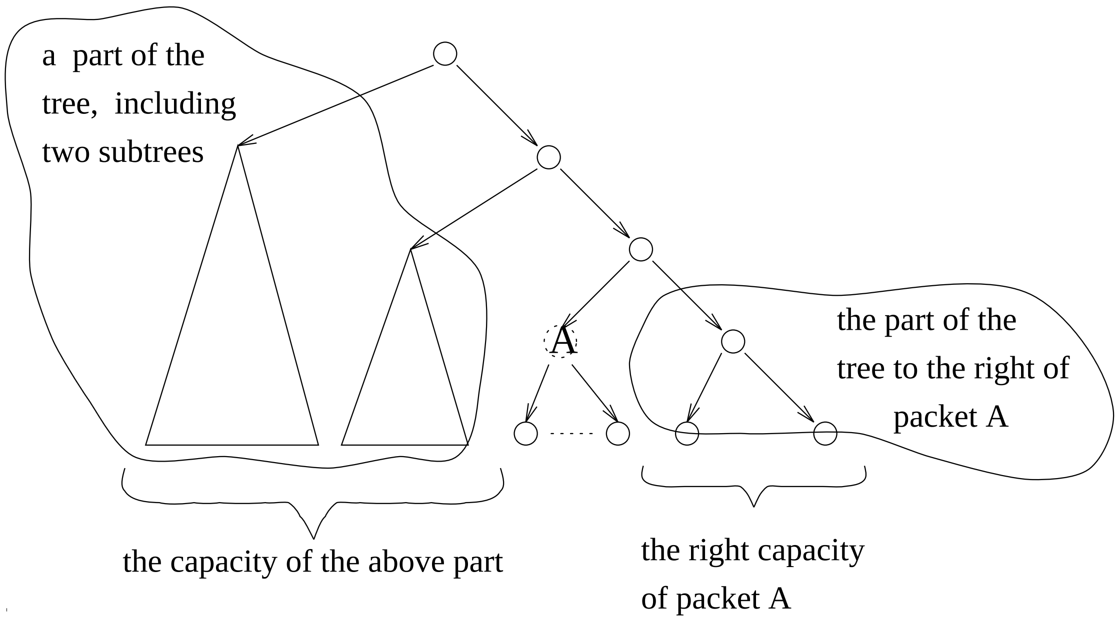

Figure 4.2

4.2 The Packet Hierarchy

The data packets, described in the last subsection, form the bottom level of a *packet hierarchy*, which is a slightly modified packet tree. In the underlying tree, the packets of level $i+1$ contain some packets of level i, as follows:

$$\begin{aligned} &P(i+1,0) = [P(i,0), P(i,1)] \\ &P(i+1,j) = \big[P(i, A(i+1,j-1)), \ldots, P(i, A(i+1,j)-1)\big], \qquad \text{for } j > 0. \end{aligned} \tag{1}$$

Thus for $i > 1$, the total size of $P(i,0), \ldots, P(i,j)$ is $A(i,j)$. It should be clear that the number of data elements that are stored in any subtree of this structure equals the total size of first level packets belonging to the subtree (its *front*). Accordingly we refer to the total size of the front of all subtrees in some part of the hierarchy as the *capacity* of that part. We define *the right capacity* $C_{i,j}$ of $P(i,j)$, where $j > 0$, to be the capacity of the part of the tree to the right of this packet, i.e. the subtrees rooted at $P(i,k)$ for $k < j$ (Figure 4.2). Our tree will have the smallest number of levels sufficient for maintaining the data stored so far, so the uppermost level consists of only one packet. We now present some of the properties of this structure:

Lemma 3. *For $i > 0$ define $C_{i,0} = 1$. Then for all $i > 0$ and $j \geq 0$, $C_{i,j} = A(i,j) - 1$.*

Proof. For $j = 0$ the statement holds by definition. For $j > 0$, We use double induction on i and j. For $i = 1$, we have

$$C_{1,j} = 2^0 + 2^1 + \ldots + 2^{j-1} = 2^j - 1 = A(1,j) - 1.$$

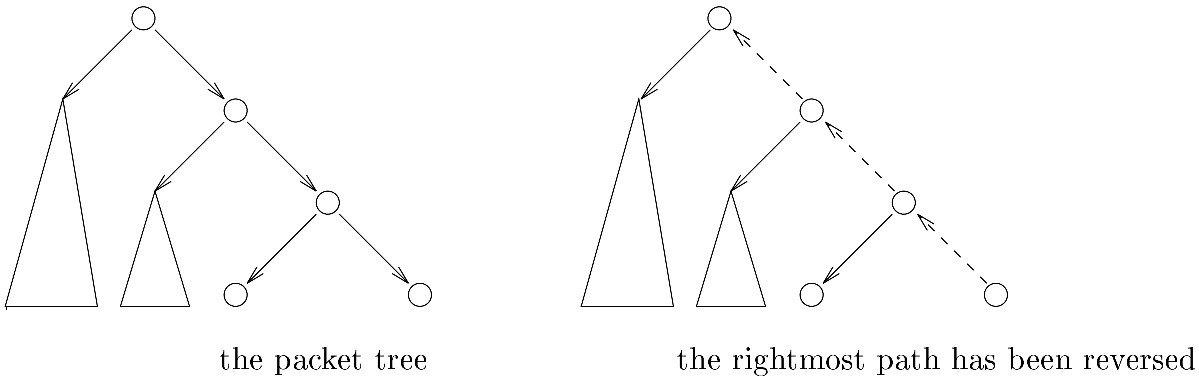

Figure 4.3

For $i > 1$, by (1) and the induction hypothesis we have

$$C_{i,j} = C_{i-1,A(i,j-1)} = A(i-1, A(i,j-1)) - 1 = A(i,j) - 1. \quad \square$$

Corollary 1. *The number of packets needed in level i for acommodating n data elements in the first level equals $\alpha_i(n)$.*

Proof. By Lemma 3, this number is $\min\{\, j > 0 \mid A(i,j) > n \,\}$. $\square$

Corollary 2. *The height of a hierarchy for storing n data elements is at most $\alpha(n) + 2$.*

Proof. We show that $\alpha(n) + 2$ levels ensure capacity of at least n. Indeed, using the fact that $A(k, 1) > k$,

$$\begin{aligned} C_{\alpha(n)+2,1} &= A(\alpha(n) + 2, 1) - 1 \\ &= A(\alpha(n) + 1, A(\alpha(n) + 2, 0)) - 1 \\ &= A(\alpha(n) + 1, 2) - 1 \\ &= A(\alpha(n), A(\alpha(n) + 1, 1)) - 1 \\ &> A(\alpha(n), \alpha(n)) - 1 > n - 1. \quad \square \end{aligned}$$

Corollary 3. *The size of $P(i,j)$, for $j > 0$, is $C_{i,j} - C_{i,j-1}$.*

Proof. Immediate from Lemma 3 and (1). $\square$

To facilitate access to our tree structure we modify it by reversing the direction of the rightmost path and storing its constituents in a stack (Figure 4.3). Thus the rightmost node of each level, $P(i, 0)$, is not stored inside $P(i + 1, 0)$. Instead of folding entirely into the topmost packet, our hierarchy resides inside a stack containing (in top down order) $P(1, 0), P(2, 0), \ldots, P(k, 0)$ where k is the current height of the tree. The place in $P(i+1, 0)$ which was to contain $P(i, 0)$ is unused. Now, to locate a node of the tree, we do not start from the root, but from the top of the stack, i.e. the rightmost node of level one. If the node we seek if farther to the left, we go up the rightmost path (down the stack) until we hit an ancestor of our target node. We then descend that partial tree.

The stack at the top of our hierarchy will be denoted by $Hier$.

$h \leftarrow \alpha_1(n - (n \wedge i)) - 1$
$P \leftarrow (Path[h], h)$ (add the block size to form a packet)
$k \leftarrow P[0]$
$H \leftarrow Hier$
for $j \leftarrow 1, \ldots, k-1$
    pop(H)
end for
$Q \leftarrow$ pop(H)
for $j \leftarrow k, \ldots, 2$
    $Q \leftarrow Q[P[j]]$
end for
return $Q[(2^h - 1) \wedge i]$

Figure 4.4
load

4.3 Auxiliary Data Structures

Apart from the packet hierarchy, three *tables* are maintained by our algorithms, and two global variables: a counter of the STOREs, called n, and the current height of the hierarchy, K. One table is called $Tail$ and will be only used in the multitasking version. The second stores $\alpha_1(i)$ for $0 \leq i < n$. Due to the existence of this table we can retrieve base 2 logarithms of numbers less than n in constant time. To keep this table current, the *store* procedure has to add $\alpha_1(n-1)$. This is done in constant time using the recurrence:

$$\alpha_1(0) = 0$$
$$\alpha_1(m) = \begin{cases} \alpha_1(m-1) & \text{if } 2^{\alpha_1(m-1)} > m \\ \alpha_1(m-1)+1 & \text{otherwise} \end{cases}$$

(we compute 2^x by shift$(1, x)$).

The third table is called $Path$. It contains, for each first level packet in existence, a packet describing the path to that packet. More specifically, consider a packet $P(1, i)$. Let its index in its parent packet be i_1, the index of the latter in its own parent i_2, and so on until we get to a rightmost packet $P(k, 0)$ (that is in the stack). It is easy to see that none of these indices can be larger than i, whence i is a valid block size for the packet, and we use it indeed and free ourselves of the need to keep the block size. In the first member of this packet we store k, which is also bounded by i. The next members are $i_1, \ldots, i_{k-1}$. The number of bits required for storing this packet is $ki \leq i^2 \leq 2^{i+1}$. This shows that it fits into the table.

4.4 LOAD

The procedure *load*, shown in Figure 4.4, accepts i as argument, and retrieves a_i. As shown in Lemma 2, we can directly compute the index h of the first level packet containing the sought datum, namely $P(1, h)$. After computing h we can obtain, from $Path[h]$, a packet P that describes the path to $P(1, h)$. $P[0]$ is the level in the hierarchy stack where an ancestor of $P(1, h)$ resides. After the loop of pop's, we obtain in Q the

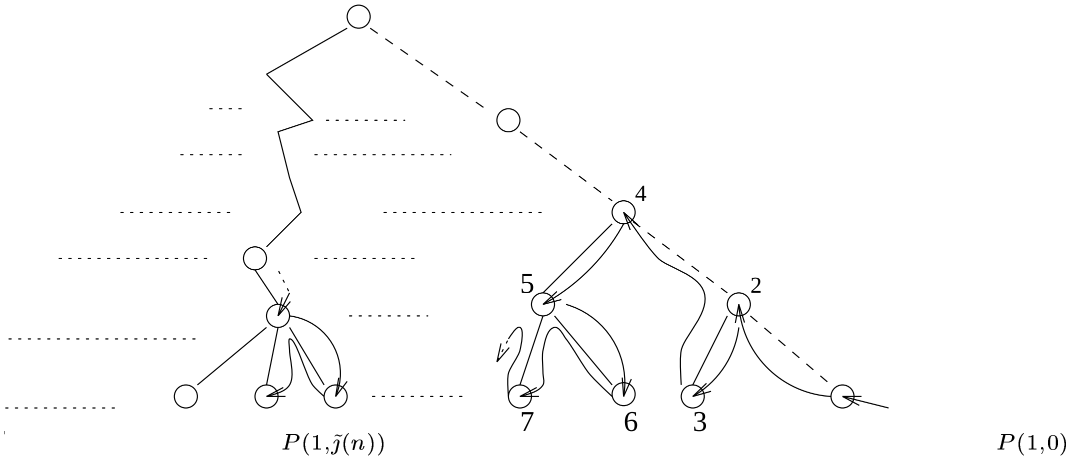

Figure 4.5
the beginning and the end of an accumulation path
the numbering shows processing sequence.

ancestor, which, by definition of packet structures contains inside it all its descendants. The second loop in *load* walks down the tree as instructed by P. At its end Q becomes $P(1,h)$ and a_i is retrieved thereof using the relation proved in Lemma 2.

All the lines in the procedure take constant time, and some are iterated $k-1$ times. k is bounded by the height of the hierarchy and by Corollary 2 k is $O(\alpha(n))$.

4.5 STORE

The *store* procedure implements the accumulation process described in Subsection 4.1. The procedure accesses $P(1,0), P(1,1), \ldots, P(1,\tilde{\jmath}(n))$ in this order (the last one may be actually created in the process). According to Section 3.5, we have to *scale* the parent of a packet that we wish to change before we can put in a new value, in order to ensure a sufficient block size. Therefore *store* processes nodes in *preorder*, defined by the following instructions (cf. Figure 4.5):
— Process $P(1,0)$, at the top of the stack.
— after processing a node, process its children in order.
— when done with a node and its descendants go up to its parent. On the rightmost path, we move to the parent in the underlying tree and process it.

This traversal of the hierarchy stops once $P(1,\tilde{\jmath}(n))$ has been processed. The path followed in the process is called *the accumulation path*, and is depicted in Figure 4.5.

Note that an accumulation visits packets in nondecreasing order of right capacities. It stops in the first leaf packet whose right capacity is at least $2^{\tilde{\jmath}(n)}-1$, where it stores the new datum. Whenever this is the first datum to enter that packet, this packet will be initialized by this accumulation, and so will its parent if the former is its first (rightmost) child. Packets with lesser right capacity must have been initialized before. We conclude that when an accumulation whose head is $\tilde{\jmath}(n)$ takes place, all the packets on the accumulation path already exist, except possibly some packets whose right capacity is n. These packets, if any, form a contiguous suffix of the path.

The algorithm maintains a fixed set of state variables, which indicate the current position in the hierarchy and determine the next moves.

Assume we are now storing a_n. Recall that B indicates the last (and largest) block size used for a first-level packet. Based on the value of B and the structure of the packet hierarchy we can precompute an upper bound on the block size needed in any packet of the hierarchy. We will use this value in scaling the packet, since we handle it before accessing its children and thus cannot know their exact size. Moreover, we do not have an instruction for finding out the size of a number, so even when we have the packet members we will not get a smaller block size. Let $i \geq 1$ and $j \geq 0$; on encountering $P(i+1, j)$ on the accumulation path, we compute the following block size:

$$S_i = 2^{i(\tilde{\jmath}(n)+1)} B$$

and scale $P(i+1, j)$ to this value. We first check if S_i does in fact exceed the current block size of the packet, since otherwise no scaling is required.

We now prove that the block size chosen this way suffices. We begin by observing that the capacity of the whole part of the hierarchy affected by this accumulation is exactly $2^{\tilde{\jmath}(n)+1}$. The proof proceeds by induction on i.

Basis: A level 2 (and higher) packet P stores a pair of numbers for each subordinate packet. Subordinate packets which are not modified by the accumulation obey the old block size of P. As for the effects of this accumulation, it may enlarge the aggregate of a first level packet up to the size of $2^{\tilde{\jmath}(n)} B$, for $2^{\tilde{\jmath}(n)}$ members and a block size of B. Thus $S_1 = 2^{\tilde{\jmath}(n)+1} B$ is a sufficient block size for the newly-updated members.

Induction step: A packet P of level $i+2 \geq 3$ houses packets of level $i+1$. Those packets that are not affected by the current accumulation are known to obey the old block size; the new size will be the larger of the old block size and S_{i+1} so these packets present no problem. As for packets modified, their new block size is the largest of the old one and S_i. If the old block size is preserved, then the current block size of P suffices as for unaffected packets. Otherwise, consider an affected packet $P(i+1, j)$. By Corollary 3, its size equals $C_{i+1,j} - C_{i+1,j-1} < C_{i+1,j}$. Also $C_{i+1,j} < 2^{\tilde{\jmath}(n)}$, for $P(i+1, j)$ has a descendant $P(1, \ell)$ such that $\ell \leq \tilde{\jmath}(n)$. For each of its subordinates, the packet $P(i+1, j)$ stores a pair of numbers using a block size of S_i, adding up to $2^{\tilde{\jmath}(n)} \cdot 2S_i$ as the size of its aggregate. Thus the largest of the old block size of P and $S_{i+1} = 2^{\tilde{\jmath}(n)} \cdot 2S_i$ is sufficient for updating P.

Similarly we handle the *Hier* stack. Recall that it stores the packets $P(1, 0), P(2, 0)$ and so on, up to the root of the packet tree. In the current accumulation, the stack members that have to be processed are $P(1, 0), \ldots, P(k, 0)$ where k can be obtained from $Path[\tilde{\jmath}(n)]$ (see § 4.3). In the case where $P(1, \tilde{\jmath}(n))$ is newly created, we can safely use $k = K + 1$, since we can go at most one level beyond the current depth of the stack. Consequently, we know that the largest value to which we may need to increase the block size of a stack member is S_k. We will assign this block size to every stack member we process (unless it already has a larger block size).

Recall that in the table *Path* we maintain, for each first-level packet, a directive packet that describes the path leading to the former through the hierarchy. Observe that an accumulation may add at most one first-level packet to the hierarchy, $P(1, \tilde{\jmath}(n))$. Thus it suffices to maintain, during the accumulation, a packet which describes the path to our current position, so that if the last packet is found to be new, this packet will be ready to enter *Path*. Maintenance is easy since the block size of that packet is known in advance to be $\tilde{\jmath}(n)$.

Here are the state variables of the algorithm:

k	a bound on the depth we go into the stack, as explained above
$i, j, C_{i,j}$	subscripts of the current node and its right capacity
$C_{i,j-1}$	self explanatory
Bs	the block size of the current packet
Bp	a bit-pointer to the current packet ($= (Ba, Bd)$ — see the definition of a bit-pointer)
C_1	capacity of the part of the hierarchy already processed
P	the accumulated packet (as in Figure 4.1)
ℓ	number of children of the current node already processed
S_1	a stack, built while going down the tree, including records of $(Siz, Bp, j, C_{i,j})$
S_2	a stack, built while going up the tree, including values of $C_{i,j}$.
Pth	a packet, of block size $\tilde{\jmath}(n)$.

The code for each iteration (visit of a node) is given in Appendix B. Initialization of the variables for the first node ($P(1,0)$) is straight-forward and omitted.

4.6 Running Time

A study of Appendix B reveals that the visit step takes constant time outside the call to *scale*. Therefore, we can divide the iteration into constant time steps by "cutting" between every two consecutive *scale steps* (Section 3). We call each constant time step so obtained an accumulation step.

Lemma 4. *Let $k_1, h_0, \ell \geq 0$ and $k_0 = k_1 + \ell \cdot 2^{h_0}$. Consider the total cost of the accumulations started for storing a_r, $r = k_1, \ldots, k_0 - 1$, where the cost of each accumulation is the number of accumulation steps it carries until it either terminates or completes processing of $P(1, h_0)$, if it happens to go further than this packet. Then there is a constant c such that this cost does not exceed $c\,\ell \cdot 2^{h_0}$.*

Proof. First we count the scale steps. Observing the form of accumulation paths, it is not hard to see that the packet $P(i,k)$ is on the path to $P(1,j)$ if and only if $C_{i,k} \leq C_{1,j} \iff C_{i,k} \leq 2^j - 1 \iff \log A(i,k) \leq j$. Therefore, $P(i,k)$ is scaled in an accumulation if and only if the latter's head $h \geq \log A(i,k)$. Recall that $A(i,k)$ is a power of two, and that $P(i,k)$ is affected by the accumulation for a_r if and only if r is divisible by $A(i,k)$. Fix i and k such that $A(i,k) \leq 2^{h_0}$ (we do not count the handling of larger packets). Then for *stores* number $k_1, \ldots, k_0 - 1$, $P(i,k)$ is scaled $\ell \cdot 2^{h_0} / A(i,k)$ times. The cost of scaling

$P(i,k)$ is $\log A(i,k)$. The total cost of scale steps is bounded by

$$\sum_{i=1}^{\infty}\sum_{k=1}^{\infty}\frac{\ell\cdot 2^{h_0}}{A(i,k)}\log A(i,k) = \ell\cdot 2^{h_0}\sum_{i=1}^{\infty}\sum_{k=1}^{\infty}\frac{\log A(i,k)}{A(i,k)} \le \ell\cdot 2^{h_0}\sum_{i=1}^{\infty}\sum_{m=\log A(i,1)}^{\infty}\frac{m}{2^m}$$
$$= \ell\cdot 2^{h_0}\sum_{i=1}^{\infty}\frac{\log A(i,1)+1}{\frac{1}{2}A(i,1)} \le \ell\cdot 2^{h_0}\sum_{r=1}^{\infty}\frac{r+1}{2^{r-1}} = 6\ell\cdot 2^{h_0}.$$

We now consider the rest of the steps, which are those visit steps where no packet is scaled. Such a step either moves into a packet on the accumulation path for the first time or leaves it for the rest of this accumulation. Since these amount to 2 steps per packet, the total cost can be at most two times the cost of the other steps, whence it is bounded by $12\ell\cdot 2^{h_0}$. We conclude that $c = 18$ satisfies the claim. □

Theorem 5. *The SM can perform retrieval by position with $T_{store} = O(1)$ and $T_{load} = O(\alpha(n))$, or $T_{load} = O(k)$ and $T_{store} = O(1 + \log\alpha_k(n))$, for any fixed k, where in both cases, the store time is amortized.*

Proof. Clearly, the worst case running time of *load* is proportional to the height of the hierarchy. By Corollary 2 it is $O(\alpha(n))$. The running time of *store* over the first n invocations can be shown to be linear in n by means of the last lemma: use $k_1 = 0$, $\ell = 1$ and $h_0 = \lceil \log n \rceil$. As for the second part of the theorem, consider a modification to the algorithm in which the hierarchy is limited in height to k. At the bottom of the stack, as the root of the sub-tree of height k, is a packet whose size will be $\alpha_k(n)$ after n data have been stored. The *store* procedure will scale this packet for each update to this sub-tree, leading to the second time bound. □

5. Retrieval By Position — Multitasking

In this section we show how to obtain retrieval by position in uniform delay where each operation takes $O(\alpha(n))$ time by means of multitasking. We refer to the single-tasking algorithm as the "old" one.

5.1 Solution Outline

In the old implementation, *load* already takes $O(\alpha(n))$ time. On the contrary, the accumulation processed by *store* may involve many packets.

Recall that multitasking is accomplished using a stack of *State Records*, which are packets encompassing the local data of a task. Tasks also access global data which, in our case, are the packet hierarchy. A task is initiated by initializing its local variables and creating an SR. A *scheduler round* consists of performing c steps on behalf of the stacked tasks, which are selected for execution in LIFO order.

In the new implementation, the store procedure will consist of two major steps:

1. Initiate an accumulation task for inserting the new datum. Push its SR on the stack.
2. Perform a scheduler round.

In general, we will have at a single time more than one accumulation task in progress (we call these *active*). Our task is to ensure that no problems arise due to concurrent access to the packet hierarchy. That is, an item within the hierarchy that should be updated by some task is not used by a later task (a task started later) before being updated, and local information that pertains to the hierarchy (in particular, bit pointers) is not invalidated by other tasks, or is restored after it is.

As pertaining to retrieval, an important feature of the multitasking implementation is that *ranges* of different accumulations are disjoint, a property which will appear in our *correctness proposition* (CP). Thus each packet that would be non-empty, had we used the old algorithm to make the insertions (let us call the result of which *the completed structure*), becomes the head of an accumulation that may be incomplete ($head \neq tail$). In order to access a_i, we apply Lemma 2 to find the index of the packet where it resides in the completed structure. Now it may turn out to be the head of an incomplete accumulation; in this case only will the packet not contain the member sought. We later show that in order to retrieve it we only need to know the location of the tail.

This version of the algorithm maintains the table $Tail$. For each packet $P(1, i)$ which is the head of an accumulation, $Tail[i]$ reflects its tail (CP guarantees that two accumulations never share a head). $Tail$ is initialized to zero and is updated by each accumulation once it starts. The maintenance of $Tail$ is simple enough to permit omitting its details.

To complete the description of the algorithm, we fill in the following parts:

1. How retrieval is performed, preserving the asymptotic time bound.
2. How we break each accumulation into steps, and how many of them should be performed in each round so that CP is preserved.
3. What the perils of concurrent update are and how they are overcome.

5.2 Retrieval

The retrieval procedure follows the outline above. It performs the "old" procedure once, and if the element it looks for is not where it expects, it computes the actual location with the help of $Tail$ and performs another "old" *load*. We now state the correctness proposition, followed by a lemma which completes the details of the *load* procedure.

Correctness Proposition (CP). *For $1 \le i \le k$ let h_i and t_i be the head and tail, respectively, of the ith youngest accumulation which is currently active. Then we have*

$$t_1 < h_1 < t_2 < \cdots < h_k.$$

In words, the ranges of the accumulations are disjoint and the older are farther to the left.

Lemma 5. *Assume CP. After n store operations, for any $0 \le i < n$, let $h = \alpha_1(n - (n \wedge i)) - 1$, $\ell = \alpha_1(2^h - 1 - ((2^h - 1) \wedge i)) - 1$ and $t = Tail[h]$. Then*

$$a_i = \begin{cases} P(1,h)[(2^h - 1) \wedge i] & \text{if } t = h, \\ P(1,t)[(2^{t+1} - 1) \wedge i] & \text{if } (2^h - 1) \wedge i \ge 2^h - 2^{t+1}, \\ P(1,\ell)[(2^\ell - 1) \wedge i] & \text{otherwise.} \end{cases}$$

Proof. If $Tail[h] = h$ (the first case) then the last accumulation that moved a_i is complete and a_i resides in $P(1, h)$. Its position is given by Lemma 2. If the last accumulation that moved a_i is not complete, then $Tail[h] < h$. The data elements to form $P(1, h)$ are now distributed as follows: the last 2^{t+1} of them reside in $P(1, t)$. The first $2^h - 2^{t+1}$ have not been moved yet. Lemma 2 states that the position of a_i among these elements is $(2^h - 1) \wedge i$. Thus if $(2^h - 1) \wedge i \geq 2^h - 2^{t+1}$, a_i will be found in $P(1, t)$, and its position within is easily seen to be $(2^{t+1} - 1) \wedge i$. Otherwise this accumulation has not moved a_i. Therefore we can use Lemma 2, replacing n by $2^h - 1$ and i by $(2^h - 1) \wedge i$ (we pretend that only the elements which are now right of $P(1, h)$ have ever been inserted). This gives the result of the lemma. □

5.3 Accumulation Tasks

The implementation of *store* in the previous section is almost ripe for multitasking. A fixed set of state variables is defined and the algorithm is presented as repeated iteration of a basic step (visit of a node). Furthermore, we showed a natural division of the program into steps of bounded duration (accumulation steps). To the local variables listed in the previous section we should add n, the number of data already stored, K, the current height of the hierarchy, and B, the current block size, since the global values may change while the accumulation is in progress.

For the number of steps to be performed in each scheduler round we use the constant c of Lemma 4.

Recall that an accumulation accesses nodes of the packet hierarchy in preorder. For convenience we relate to the packets in the hierarchy as ordered this way.

5.4 Proof of CP

We prove CP by induction on time, beginning just before the first operation, where no task is started and the proposition trivially holds. We continue to prove the proposition for *store* k_0 (clearly, only *store*s affect the proposition). *Store* k_0 enters a_{k_0} into memory, and is actually the $k_0 + 1$st store operation.

Assume that in all previous steps, CP was in effect. Let $h_1, \ldots, h_k$ be the accumulation heads just before this step. Let the head of the accumulation started by this *store* be called h_0. We informally use the head of an accumulation as a name for that accumulation. A violation of CP takes place only if $h_0 \geq t_1$. Assume that this is the case. Let k_1 denote the store in which h_1 was started. An accumulation, whose head is h, is started when storing a_n only if n is a multiple of 2^h. Therefore k_1 is a multiple of 2^{h_1} and k_0 a multiple of 2^{h_0}. Let $h_{\min} = \min(h_1, h_0)$. Then $k_0 - k_1$ is a multiple of $2^{h_{\min}}$. During steps $k_1, \ldots, k_0 - 1$ no step was performed on behalf of accumulations older than h_1, since h_1 has to be completed first. Also, none of the accumulations started had head $> h_{\min}$, since then CP would have been violated at that time. It follows that the budget of $(k_0 - k_1)c$ steps allowed by the scheduling strategy was used up by the accumulations started in *loads* $k_1, k_1 + 1, \ldots, k_0 - 1$. By Lemma 4, this budget suffices to bring the tail of the first accumulation (h_1) past $h_{\min}$—a contradiction. □

5.5 Overcoming Concurrency Problems

Whenever a task is suspended before a scale step, it holds in its local variables a copy of a packet being scaled. If this packet would be modified by another task concurrently disaster will ensue. This, however, is prevented by CP: consider the task currently in execution. It does not modify any packets past its head (in the preorder defined above), while suspended tasks have already passed it. Thus there is no concurrent update of a packet. But younger tasks do scale packets that are ancestors of the current node of older, active, tasks. They may also change the stack. Since handling the stack is similar to that of a packet, we do not discuss it separately. Such behavior of the young causes the current node of older tasks, or an ancestor thereof, to change its bit position and enclosing block size, which the old algorithm keeps in Bs, Bp and S_1.

In order to solve this problem we avoid keeping this kind of information in the local memory of any task. Thus we remove the variables Bs and Bp (as well as their records in S_1). We do not need these variables during the scale steps. In steps where we need them, we will recover the current values from the hierarhy; since we know the path from the top of the hierarchy down to the current packet (it is recorded in Pth), we can follow it (much as in *load*), maintaining at the same time a bit pointer into the hierarchy, which upon reaching level i will display the value for Bp and allow us to retrieve Bs (still, as in *load*). Stopping at level $i+1$, we obtain the value of Bd at that level, instead of getting it from S_1. The cost of this process is $O(\alpha(n))$ time, just as for *load*.

This completes the description of the algorithm, as well as the clarification of its correctness and complexity, as summarized in the following theorem.

Theorem 6. *The SM can perform retrieval by position in $O(\alpha(n))$ time per operation.*

6. From Retrieval by Position to RASUS

The time bounds claimed for the RASUS problem in Theorems 1 and 2 are the same as we obtained for retrieval by position. Therefore, we can prove the former by "reducing" the problem of RASUS to retrieval by position (henceforth RP) while maintaining the complexity.

Two difficulties in RASUS separate it from RP. The first is that each STORE specifies an arbitrary address by which the adjoining datum must be retrieved. The second is multiple stores into the same address. If we do not handle these appropriately we may end up storing into our RP structure $n \gg s$ items (s is the number of distinct addresses in theorems 1 and 2). This causes the running time of the RP operations to depend on n.

The solution is based on an RP structure augmented with an *address map* to translate RASUS addresses to RP indices. This mechanism works as follows: let n be a counter of the *stores*. $\mathrm{store}(a, d)$ stores a and then d into the RP structure (into positions $2n$ and $2n+1$). This is performed by the *store* procedure for RP, which we denote by $RP.\mathrm{store}()$. Let the map be called $Addr$; n is incremented by one and then stored into $Addr[a]$. Thus, to perform $\mathrm{load}(a)$, we call $RP.\mathrm{load}(2 \cdot Addr[a] - 1)$. Some observations about this mechanism:

(i) the time for *store* and *load* is proportional to the time of RP for the same value of n.

(ii) if, before performing store(a, d), $Addr[a]$ is nonzero, we are storing into a used address, making some former datum in the RP structure obsolete. We can check this in the *store* procedure and maintain the counter s of distinct addresses used. Since a is stored in the RP structure just before d, it can be determined if the datum at a given position j is obsolete by checking whether $Addr[RP.\mathrm{load}(j-1)] = \frac{j-1}{2} + 1$.

(iii) we have not solved the whole problem yet. First, the numbers stored in $Addr$ grow indefinitely, enforcing a growth in the block size. Second, n may grow arbitrarily larger than s, and our running time is still a function of n.

The solution of the two problems lies in refreshing our data structure repeatedly: we will maintain two structures in the above manner. Following every store operation, the current structure CUR is guaranteed to contain all the data and is used by *load*; while NEW, the fresh one, is still under construction and will replace CUR in its turn, completing a *round*. We distinguish between the structures in notation by prefixing their name to an operation, e.g. CUR. load, pertaining to the RP structure in CUR, or to a variable, e.g. NEW.b, the block size of the address map in NEW.

We now describe a typical round. It begins in a situation where CUR.$n \leq 2s$ and CUR.$n < 2^{\mathrm{CUR}.b-1}$. We initialize NEW with NEW.b = CUR.$b + 1$. Now every store operation first updates CUR in the manner described above, and then, traversing the RP structure, gets the next three (a, d) pairs from CUR and enters those of them that are not obsolete in NEW. We later show that these load operations do not lengthen the time of the store operation by more than a constant factor.

Let n_1 be the value of CUR.n at the start of the round. We can safely assume that n_1 is even. Let s_1 be the number of distinct addresses stored up to that moment in CUR. Thus $n_1 \leq 2s_1$. It is easy to see that after $\frac{n_1}{2}$ store operations, all the significant data will have been moved to NEW. The store operations applied to NEW may amount to $\frac{n_1}{2}$ for the data added in this round, while out of the older n_1 data, only s_1 will be stored. Thus, at the end of the round, we have NEW.$n \leq s_1 + \frac{n_1}{2} \leq 2s_1 \leq 2s$. Moreover, NEW.$n \leq n_1 + \frac{n_1}{2} \leq 2n_1 \leq 2 \cdot 2^{\mathrm{CUR}.b-1} = 2^{\mathrm{NEW}.b-1}$. So NEW is ready to replace CUR for the next round. Similarly, during the whole round, CUR.$n < 2^{\mathrm{CUR}.b}$. This guarantees that the block size of CUR.$Addr$ suffices. Also CUR.$n \leq 3s$, whence the asymptotic running time is the desired function of s.

We still have to justify the claim on the running time of the *load* operations done for transferring the contents of CUR to NEW. For the RP variants where *store* costs more than *load*, the statement is immediate. The only exception is the single-tasking algorithm, which achieves *store* in $O(1)$ amortized time, while *load* requires $O(\alpha(n))$ steps. In this case, we get the total time of these loads to be proportional to n, exploiting the fact that these load operations constitute an orderly traversal of the structure. We simply keep each first level packet accessed in a dedicated register until all the data therein have been retrieved. Recall that n input data occupy $\log n$ packets in the first level; Thus we get an amortized running time of $\log n \times O(\alpha(n))$ for fetching these packets, and $O(n)$ for retrieving the values stored therein, for a total of $O(n)$.

7. Space Complexity

It is clear that in our methods of packing whole data structures into a few registers, not all the bits of these registers contain data: first, some space is used for values (such as block sizes), which are not the problem's data. Second, some space must be wasted since, to enable fast retrieval, we attempt at placing data in regularly spaced positions. This is complicated by the fact that the size of the input numbers is not known in advance. To find out the storage overhead involved, we study the length of the integers produced in the simulation given a bound w on the word length of the simulated RAM. Note that the memory size of this RAM is Aw bits, where A is the largest address is use.

Let B be the largest block size used while storing n data into the *retrieval by position* structure of Section 4. Then the *store* procedure calculates the block size for level i in the packet hierarchy to be $2^{i(\lceil\log n\rceil+1)}B$. It follows by Corollary 2 that the block size for the topmost packet is at most $2^{(\alpha(n)+2)(\lceil\log n\rceil+1)}B$.

Now, this block size is used for every member of the main stack, which contains one for each level of the hierarchy; whence the size of the aggregate word of this stack equals (up to constant factors) $\alpha(n)2^{(\alpha(n)+2)(\lceil\log n\rceil+1)}B$.

Clearly, the registers holding the block size for the stack and the global variables n, K and B require fewer bits than the above. The same goes for the tables α_1 and $Path$, because each of them uses $O(n)$ bits. We obtain:

Theorem 7. *Single-tasking retrieval by position uses $O(\alpha(n)2^{(\alpha(n)+2)(\lceil\log n\rceil+1)}B)$ bits for storing n numbers with largest block size B.*

We see that the data block size B is multiplied by an expression dependent on n, which is just above polynomial, belonging to the class $\dot{O}(n^{\alpha(n)})$ (the "dotted O" notation is from Ben-Amram (1991). $f = \dot{O}(g)$ if $f(n)$ is dominated by $g(cn)$ for some constant c). The block size itself will be taken by our algorithm to be the largest input number, possibly as large as 2^w. This is due to the fact that there is no direct way of determining the size of an arbitrary number. Hence if w is large (relative to n), much space is wasted. However, in realistic RAM programs, it is reasonable to assume $w \leq n$; then a table of size n can furnish us with the exact size of each input number. In this case, the block size B will be the exact size of the number and incur no waste (it may incur a saving).

Let us consider now the multitasking algorithm. The difference from the former version is in the addition of the table $Tail$, whose size is $O(n)$ bits, and the SR stack. This stack contains a record for each task. The largest members of this record are the packet P and local variables of the *scale* procedure (a, m) whose size matches the size of the scaled packet. For a task whose head is h, we determine (by the same arguments as above) the size of the largest packet involved to be $2^{(\alpha(n)+2)(h+1)}B$. This size can be calculated in the course of the algorithm and used for the block size of the SR packet. The total size of the packet will be proportional to this expression. Different tasks must have different heads; assuming we assign block sizes cleverly based on the above expression, we obtain for the total size of this stack

$$\sum_{h=0}^{\lfloor \log n \rfloor} 2^{(\alpha(n)+2)(h+1)} B = O(2^{(\alpha(n)+2)(\lfloor \log n \rfloor + 1)} B).$$

Theorem 8. *The multitasking retrieval by position algorithm uses the same space (ignoring constant factors) as the single-tasking one, namely $O(\alpha(n) 2^{(\alpha(n)+2)(\lceil \log n \rceil + 1)} B)$ bits for storing n numbers with largest block size B.*

Finally we consider the full solution to the RASUS problem. The algorithm maintains two *retrieval by position* data structures, both conforming to the above bounds for $n \le 2s$. So it relates to s and B as discussed above for n and B. In addition the map $Addr$ is maintained. It contains an entry for each address in use, storing an index between 0 and $n \le 2s$. Thus the size of the map in bits is $O(A \log s)$. Assuming $\log s \le w$ (a natural assumption, since addresses normally fit within a word), this is bounded by $O(Aw)$ (the memory size of the simulated RAM).

We conclude by noting that if the memory usage in the RAM is sparse ($A = \dot{\Omega}(s^{\alpha(s)})$), and $w \le s \le 2^w$, then the number of bits in the SM registers is bounded (up to a multiplicative constant) by the memory size of the RAM, Aw. If the memory usage in the RAM is not sparse, the simulation multiplies the space by $\dot{O}(s^{\alpha(s)})$.

8. Conclusion

The effects of varying the instruction set on the power of a computational model (notably the RAM) is a long standing issue in computer science. The contribution of this paper is in assessing the power of the shift instruction, under a model that allows an arbitrary shift in unit cost. Ours is the first study of this problem in the context of high level models of computation, which deal naturally with domain values (in contrast with bits). In such a framework, it was shown by Ben-Amram and Galil (1992) that many and powerful instruction sets possess the property of *incompressibility*, which states that *memory* is a valuable resource: it cannot be traded for any bounded amount of time. On the contrary, the Shifting Machine can contract the whole memory contents of the RAM into a fixed number of registers, and not pay much — the RAM is simulated almost in real time. Thus this instruction set crosses some barrier which the instruction sets discussed in (Ben-Amram and Galil, 1992) respect. The result indicates that the primitive of unbounded shift is almost as strong as the mechanism of random access to unbounded memory. We conjecture that the gap was narrowed in this paper to a minimum; at least, a linear RASUS for the SM does not exist. Proving this is an open problem. However, it is crucial to note that the intricacy of our simulation stems from the fact the the size of numbers stored is not known in advance. If a bound w on the size of the numbers were known in advance, RASUS would be solved by a single *map*, containing Aw bits. There is no storage overhead and each access to the structure takes constant time. Thus in the many situations where such a bound is known, *shift* simulates random addressing in real time.

As in previous work (Pratt and Stockmeyer, 1976; Schönhage, 1979; Simon, 1981; Bertoni, Mauri and Sabadini, 1981) we are interested in cutting down the instruction set needed for our simulation. The candidates for removal are multiplication and the AND instruction.

Multiplication is only used in our algorithms for the purpose of indexing a packet, where the location of member i is computed as ib. Without a multiply instruction, the shift instruction may be used to effect multiplication by a power of two. To this end, a packet represented as (a, b) will use the block size 2^b, and so the position of member i will be obtained by $\text{shift}(i, b)$. The cost of this change is a large increase in the size of numbers created, hence the bit-level space bounds derived in the last section will not be obtained. Yet the simulation time remains the same.

The Boolean instruction AND is used in a more essential manner, which does not seem to lend itself to direct replacement by other instructions as above. Nevertheless, we can obtain simulation in $O(\alpha(s))$ time per operation with somewhat different structures. We will spare the reader these details.

How should we interpret the astounding power of the shifting model? Pratt and Stockmeyer (1976) indicate two possible directions. On one hand, this model may be considered unrealistic; our results sustain this conclusion, since unit-cost random addressing of an unbounded memory is considered an unrealistic feature of the RAM (Aggarwal et al., 1987; Tarjan, 1977). On the other hand, they suggest that a parallel architecture may implement bit-vector operations (i.e. operations on very long binary integers) efficiently enough to make the fast algorithms relying on this model usable (they give examples of such algorithms). Our results indicate that such a device cannot be essentially faster than a similar implementation of a large random access memory.

A. Proof of Claim 1

In this appendix we complete the proof of Claim 1, by proving the correctness of procedure *scale* (Figure 3.1). First we introduce some notation. Let $0 \le x_i < 2^b$ and $a, y_i \in \mathbb{N}$. We write $\bigvee_i (x_i \ll y_i)$ for $\sum_i x_i \cdot 2^{y_i}$ provided that for all i, $\mathrm{Mid}_{y_i,b}(a) = x_i$ (b is to be understood from context). It is easy to see that $a = \bigvee_i (x_i \ll y_i)$ must have all the "unused" bits zero, i.e. for all i such that $\forall j : i \notin [y_j, y_j + b - 1]$ we have $\mathrm{Mid}_{i,1}(a) = 0$. Using this notation, a packet $P^{(n)} = (a, b)$ satisfies $a = \bigvee_{i=0}^{n-1} (P[i] \ll ib)$.

Lemma 6. *Let $u = \bigvee_{i \in S} (x_i \ll y_i)$ and $v = \bigvee_{j \in T} (x_j \ll y_j)$, where $S \cap T = \emptyset$.*

(i) If for all $i \in S$ and $j \in T$ we have $[y_i, y_i + b - 1] \cap [y_j, y_j + b - 1] = \emptyset$ then $u + v = \bigvee_{k \in S \cup T} (x_k \ll y_k)$.

(ii) If $\{ (x_j, y_j) : j \in T \} \subseteq \{ (x_i, y_i) : i \in S \}$ then $u - v = \bigvee_{i \in S - T} (x_i \ll y_i)$.

Proof. (i) $u + v = \sum_{i \in S} x_i \cdot 2^{y_i} + \sum_{j \in T} x_j \cdot 2^{y_j} = \sum_{i \in S \cup T} x_i \cdot 2^{y_i}$. Also, by the disjointness of the intervals $[y_i, y_i + b - 1]$, $i \in S \cup T$ it follows that for each such i, $\mathrm{Mid}_{y_i,b}(u + v) = \mathrm{Mid}_{y_i,b}(u) + \mathrm{Mid}_{y_i,b}(v) = x_i$. (ii) follows directly from (i). □

Let us repeat the definition of the *move* procedure.

$$
\begin{array}{l}
\mathrm{move}(a, m, k)\textbf{:} \\
\quad t \leftarrow a \wedge m \\
\quad a \leftarrow a - t \\
\quad a \leftarrow a + \mathrm{shift}(t, k)
\end{array}
$$

Lemma 7. *Let $a = \bigvee_{i=1}^{n} (x_i \ll y_i)$, where $x_i < 2^b$. Let $S \subseteq \{1, \ldots, n\}$, $m = \bigvee_{i \in S} (2^b - 1 \ll y_i)$, and $k \in \mathbb{N}$ such that for all $i \in S$ and $j \notin S$, $[y_i + k, y_i + k + b - 1] \cap [y_j, y_j + b - 1] = \emptyset$. Then executing* $\mathrm{move}(a, m, k)$ *results in $a = \bigvee_{i=1}^{n} (x_i \ll y_i + k\chi_S(i))$, where $\chi_S(i)$ is 1 if $i \in S$ and 0 otherwise.*

Proof. m is a mask in which bits $y_i, \ldots, y_i + b - 1$ are 1 for all y_i such that $i \in S$ and zero otherwise. Therefore, the value t gets is $\bigvee_{i \in S} (x_i \ll y_i)$ and in the next line these elements are removed from a. In the third line, they are put back in their new places (Lemma 6 justifies these statements). □

We can now prove the correctness of procedure *scale*. Since adding zeros to a packet does not change it, w.l.o.g. assume n to be a whole power of 2. We prove that whenever the **while** condition is tested the following equalities hold:

$$
\text{(i)} \quad a = \bigvee_{i=0}^{n-1} \big(P[i] \ll id - \mathrm{Right}_l(i)(d - b) \big)
$$

$$
\text{(ii)} \quad m = \bigvee_{i<n : \mathrm{Mid}_{l-1,1}(i) = 1} \big(2^b - 1 \ll id - \mathrm{Right}_l(i)(d - b) \big) \qquad \text{(only for } l > 0\text{)}.
$$

These equations certainly hold for the initial values of a and m. Assume that they hold when we start some iteration of the loop. We claim that Lemma 7 applies to line 5, for

$x_i = P[i]$, $y_i = id - \mathrm{Right}_l(i)(d-b)$ and $S = \{\, i < n : \mathrm{Mid}_{l-1,1}(i) = 1 \,\}$. To verify this, we have to show that for all $i \in S$ and $j \notin S$, $[y_i + k, y_i + k + b - 1] \cap [y_j, y_j + b - 1] = \emptyset$. Let $I_i = [y_i + k, y_i + k + b - 1]$ and $J_j = [y_j, y_j + b - 1]$. It is easy to see that S contains exactly the second half of each of the intervals $(r-1)2^l \ldots r2^l - 1$, for $0 < r \leq n2^{-l}$. For each interval R denote its second half by R_2. We show that $\bigcup_{i \in R_2} I_i \cap \bigcup_{j \notin R_2} J_j = \emptyset$, which trivially entails our result. To this end, note that the smallest element of $\bigcup_{i \in R_2} I_i$ is

$$\begin{aligned}
y_{(r-1)2^l + 2^{l-1}} + k &= y_{(2r-1)2^{l-1}} + 2^{l-1}(d-b) \\
&= (2r-1)2^{l-1}d - \mathrm{Right}_l((2r-1)2^{l-1})(d-b) + 2^{l-1}(d-b) \\
&= (2r-1)2^{l-1}d.
\end{aligned}$$

Its largest element is

$$\begin{aligned}
y_{r2^l - 1} + k + b - 1 &= (r2^l - 1)d - \mathrm{Right}_l(r2^l - 1)(d-b) + 2^{l-1}(d-b) + b - 1 \\
&= (r2^l - 1)d - (2^{l-1} - 1)(d-b) + b - 1 \\
&= (2r-1)2^{l-1}d + 2^{l-1}b - 1.
\end{aligned}$$

Hence

$$\bigcup_{i \in R_2} I_i \subseteq [(2r-1)2^{l-1}d, (2r-1)2^{l-1}d + 2^{l-1}b - 1].$$

Now for $j = r2^l$, we have

$$\begin{aligned}
& y_j = r2^l d - \mathrm{Right}_l(r2^l)(d-b) = r2^l d > (2r-1)2^{l-1}d + 2^{l-1}b - 1 \\
\implies & \bigcup_{i \in R_2} I_i \cap \bigcup_{j \geq r2^l} J_j = \emptyset.
\end{aligned}$$

Similarly for $j = (r-1)2^l + 2^{l-1} - 1$ we have

$$\begin{aligned}
y_j + b - 1 &= ((r-1)2^l + 2^{l-1} - 1)d - \mathrm{Right}_l((r-1)2^l + 2^{l-1} - 1)(d-b) + b - 1 \\
&= (r-1)2^l d + 2^{l-1}b - 1 < (2r-1)2^{l-1}d \\
\implies & \bigcup_{i \in R_2} I_i \cap \bigcup_{j < (r-1)2^l + 2^{l-1}} J_j = \emptyset.
\end{aligned}$$

These two results together prove our claim.

Now the lemma states that

$$a = \bigvee_{i=0}^{n-1} \big(P[i] \ll id - \mathrm{Right}_l(i)(d-b) + k\chi_S(i)\big).$$

We have

$$k\chi_S(i) = k\mathrm{Mid}_{l-1,1}(i) = 2^{l-1}(d-b)\mathrm{Mid}_{l-1,1}(i).$$

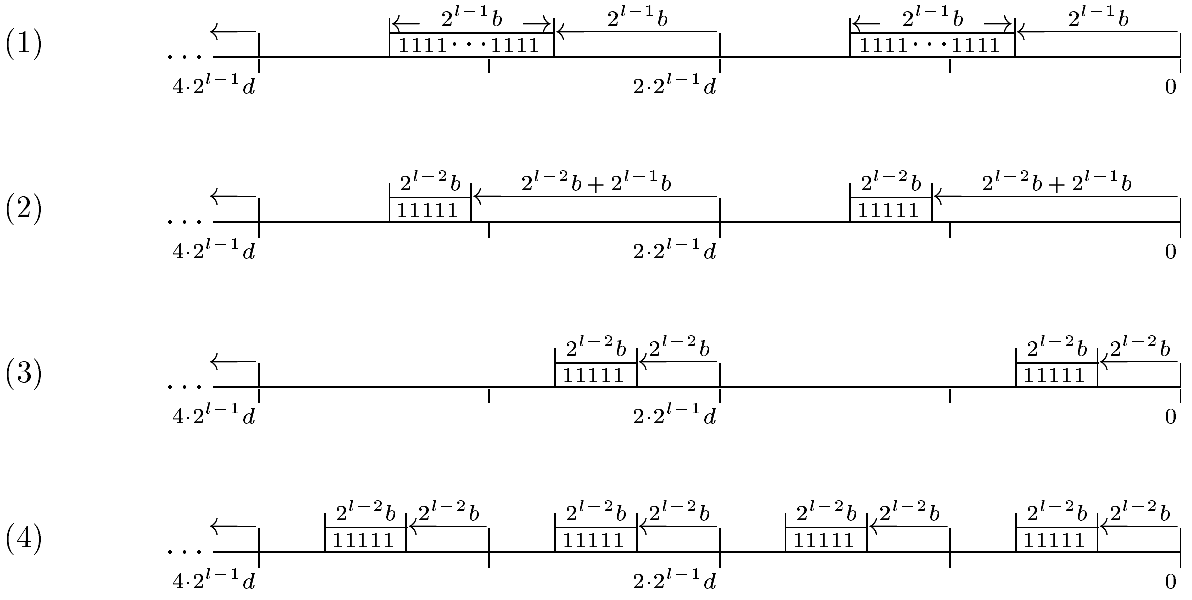

Figure A.1
a pictorial representation of the expressions (1)–(4).

Trivially,

$$\mathrm{Right}_l(i) = \mathrm{Right}_{l-1}(i) + 2^{l-1}\mathrm{Mid}_{l-1,1}(i)$$

and hence

$$id - \mathrm{Right}_l(i)(d-b) + k\chi_S(i) = id - \mathrm{Right}_{l-1}(i)(d-b).$$

This makes (i) true again after line 10. For (ii), we first note that it can be rewritten as follows:

$$m = \bigvee_{j=0}^{2^{\log n - l}-1} \left(2^{2^{l-1}b} - 1 \ll (2jd+b)2^{l-1}\right) \tag{1}$$

that is, m is composed of blocks of $2^{l-1}b$ one bits. The distance between the least-significant-bits of the blocks is $2d2^{l-1}$ and hence between two successive blocks we have a string of $(2d-b)2^{l-1} \geq 2^{l-2}b$ zeros. This means that the shift in line 7 brings zeros to the first $2^{l-2}b$ positions of each block, and after the AND we have

$$m = \bigvee_{j=0}^{2^{\log n - l}-1} \left((2^{2^{l-2}b} - 1)2^{2^{l-2}b} \ll (2jd+b)2^{l-1}\right). \tag{2}$$

The shift in line 8 produces

$$\bigvee_{j=0}^{2^{\log n - l}-1} \left((2^{2^{l-2}b} - 1)2^{2^{l-2}b} \ll (2jd)2^{l-1}\right). \tag{3}$$

It is readily verified that the blocks of the two terms do not overlap and hence by Lemma 6 we have after line 8 a mask which contains the blocks of both. By (1) and Lemma 7 it follows that the *move* in line 9 adds k to the displacements of just the blocks of (2). The disjointness condition holds because the distance between a block of (2) and the next block of (3) is $(4d-3b)2^{l-2} > (d-b)2^{l-1} = k$. A block of (2) starting at bit $s = (4jd+3b)2^{l-2}$ gets moved to $s+k = (2j+1)d2^{l-1} + b2^{l-2}$, so that finally

$$m = \bigvee_{j=0}^{2^{\log n - l+1}-1} \left(2^{2^{l-2}b} - 1 \ll (2jd+b)2^{l-2}\right) \tag{4}$$

which, after l is decremented, becomes (1) again.

When l reaches zero, (i) becomes

$$a = \bigvee_{i=0}^{n-1} \left(P[i] \ll id - \mathrm{Right}_0(i)(d-b)\right) = \bigvee_{i=0}^{n-1} \left(P[i] \ll id\right). \qquad \square$$

B. Code for an iteration of Algorithm 3

The following routine implements one iteration of the single tasking *store* routine, in which a node is processed and then a move is made to the next node in preorder (as described in Section 4.4). In order to understand this code, refer to the variable definitions in Section 4. In the code we use C-like notation in which given a pointer Bp to a packet, $Bp[i]$ references a particular member, $*Bp$ the packet itself, and $Bp{\to}a$, $Bp{\to}b$ its two constituents. If Bp points to a stack member, $Bp{\to}$next is the bit-pointer to the following member. Finally, "&P" forms a bit-pointer to P.

```
if i = 1 then
    (handling a first level packet, see Section 4.1)
    if j = j̃(n) then
        *Bp ← P
        Path[j̃(n)] ← Pth
        stop (end of accumulation)
    else
        Q ← *Bp
        scale(Q, B)
        P ← combine(P, Q)
        *Bp ← P
        C_1 ← 2C_1 + 1
    end if
else
    (visiting an internal packet)
    if ℓ = 0 then
```

(first time at this node, scale it upon need)
if $Bs < 2^{i(\tilde{\jmath}(n)+1)}B$ **then**
$Bs \leftarrow 2^{i(\tilde{\jmath}(n)+1)}B$
scale($*Bp, Bs$)
endif
endif
if $\ell < Siz$ **then**
(move on to next child)
push($(Siz, Bp, j, C_{i,j}), S_1$)
$Bp \leftarrow \&(Bp[\ell])$
$Bs \leftarrow Bp {\rightarrow} b$
$i \leftarrow i - 1$
$j \leftarrow C_{i,j-1} + \ell + 1$
$C_{i,j} \leftarrow C_1$
$C_{i,j-1} \leftarrow$ pop(S_2)
$Pth[i] \leftarrow \ell$
$\ell \leftarrow 0$
return (end of iteration)
end if
end if

– this point is reached when we are done with a node and have to go up.
$i \leftarrow i + 1$
if $j = 0$ **then**
(we are on the rightmost path, namely the stack)
$Pth[0] \leftarrow i$
$d \leftarrow (Bp {\rightarrow} \text{next} {\rightarrow} b)$
if $d = 0$ **then**
$K \leftarrow K + 1$
(we fell off the end of the stack)
$d \leftarrow 2^{k(\tilde{\jmath}(n)+1)}B$
$(Bp {\rightarrow} \text{next} {\rightarrow} b) \leftarrow d$
else if $d < 2^{k(\tilde{\jmath}(n)+1)}B$ **then**
$d \leftarrow 2^{k(\tilde{\jmath}(n)+1)}B$
adjust the stack to have $Bp {\rightarrow} \text{next} {\rightarrow} b = d$
end if
$Bp \leftarrow (Bp {\rightarrow} \text{next})$
$Bs \leftarrow (Bp {\rightarrow} a {\rightarrow} b)$ (the block size of the packet, stored in the stack at Bp)
$\ell \leftarrow 1$
push($C_{i,j}, S_2$)
else
(returning to a parent)
$\ell \leftarrow j$
$Bs \leftarrow Bd$
$(Bp, j, C_{i,j}, C_{i,j-1}) \leftarrow$ pop(S_1)

$\ell \leftarrow \ell - C_{i,j-1}$
push$(C_{i,j}, S_2)$
end if
return (end of iteration)

Acknowledgments

The authors wish to thank David Eppstein for pointing out that the time bounds we achieved can be described by the well known α functions. Section 7 was added to answer a question raised by the editor, Larry Stockmeyer.

REFERENCES

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

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

Ben-Amram, A. M. (1988), *On Pointers versus Addreses.* MSc Thesis, Department of Computer Science, Tel-Aviv University.

Ben-Amram, A. M. (1991), "Some notions on notations," *SIGACT News* 22:4, 60–62.

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

Bertoni, A., Mauri, G. and Sabadini, N. (1981), "A characterization of the class of functions computable in polynomial time on random acess machines," *Proc. 13th annual ACM Symp. on Theory of Computing*, Milwaukee, Wisconsin.

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

Hart, S. and Sharir, M. (1986), "Nonlinearity of Davenport-Schinzel sequences and of generalized path compression schemes," *Combinatorica* 6, 151–177.

Hartmanis, J. and Simon, J. (1974), "On the power of multiplication in random access machines," *15th Annual Symp. on Switching and Automata Theory*, 13–23.

R. Péter (1967), *Recursive Functions.* I. Földes (tr.), Academic Press and Akadémiai Kiadó, Budapest.

Pratt, V. R., Rabin, M. O. and Stockmeyer, L. J. (1974), "A characterization of the power of vector machines," *Proc. 6th annual ACM Symp. on Theory of Computing*, Seattle, Washington.

Pratt, V. R. and Stockmeyer, L. J. (1976), "A characterization of the power of vector machines," *J. Comput. System Sci.* 12, 198–221.

Schönhage, A. (1979), "On the power of random access machines," *Proc. 6th ICALP (1979),* LNCS 71, Springer, Berlin.

Simon, J. (1977), "On feasible numbers," *Proc. 9th annual ACM Symp. on Theory of Computing*, Boulder, Colorado.

Simon, J. (1981), "Division in unit cost RAMs," *J. Comput. System Sci.* 22, 421–441.

Simon, J. and Szegedy, M. (1992), "On the coplexity of RAM with various operation sets," *Proc. 24th annual ACM Symp. on Theory of Computing*, Victoria, Canada.

Tarjan, R. E. (1977), "Reference machines require nonlinear time to maintain disjoint sets," *Proc. 9th annual ACM Symp. on Theory of Computing*, Boulder, Colorado.

Tarjan, R. E. (1983), *Data Structures and Network Algorithms,* Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania.