Math. Systems Theory 25, 141–159 (1992)

Mathematical Systems Theory

On the Time and Space Complexity of Computation Using Write-Once Memory*
or
Is Pen Really Much Worse than Pencil?

Sandy Irani,[1] Moni Naor,[2] and Ronitt Rubinfeld[1]

[1] Computer Science Division, University of California, Berkeley, CA 94720, USA

[2] IBM Almaden Research Center, San Jose, CA, USA

Abstract. We introduce a model of computation based on the use of write-once memory. Write-once memory has the property that bits may be set but not reset. Our model consists of a RAM with a small amount of regular memory (such as logarithmic or n^{α} for $\alpha < 1$, where n is the size of the problem) and a polynomial amount of write-once memory. Bounds are given on the time required to simulate on write-once memory algorithms which originally run on a RAM with a polynomial amount of regular memory. We attempt to characterize algorithms that can be simulated on our write-once memory model with very little slow-down. A persistent computation is one in which, at all times, the memory state of the computation at any previous point in time can be reconstructed. We show that any data structure or computation implemented on this write-once memory model can be made persistent without sacrificing much in the way of running time or space. The space requirements of algorithms running on the write-once model are studied. We show that general simulations of algorithms originally running on a RAM with regular memory by algorithms running on our write-once memory model require space proportional to the number of steps simulated. In order to study the space complexity further, we define an analogue of the pebbling game, called the pebble-sticker game. A sticker is different from a pebble in that it

* The research of S. Irani was supported by NSF Grant No. DCF-85-13926, and a Tandem Corporation Fellowship. R. Rubinfeld's research was supported by NSF Grant No. CCR 88-13632 and an IBM Graduate Fellowship.

cannot be removed once placed on a node of the computation graph. As placing pebbles correspond to writes to regular memory, placing stickers correspond to writes to the write-once memory. Bounds are shown on pebble-sticker tradeoffs required to evaluate trees and planar graphs. Finally, we define the complexity class WO-PSPACE as the class of problems which can be solved with a polynomial amount of write-once memory, and show that it is equal to P.

1. Introduction

Write-once memory is memory where bits may only be "used" once, in that they can be set but not reset. This is the same difficulty we encounter when using pen rather than pencil and eraser. The write-once property is interesting for several reasons:

1. It occurs naturally in more and more sophisticated technologies from stone tablets to punch cards and paper tapes to optical disks.
2. It may be a good way to view memory where the time required to erase is much greater than the read/write time so that we would not want to erase during the computation. This property occurs in the optical disk of the NEXT computer.
3. It models the restriction in PROLOG that allows variables to be bound only once [PMN].
4. Implementations of parallel operations on data structures are made difficult by synchronization issues. I-structures, which use write-once restrictions, have been introduced as a simple and elegant way to enforce synchronization [ANP].

We investigate the implications that the write-once property has on both the time and space requirements of computation and we show that, in fact, write-once memory can be used very effectively, though not always without some loss in time and space.

Previous work in this area has been done by Vitter with the emphasis on the use of optical disks in databases [Vi2]. Rivest and Shamir give codes which allow making updates to a variable using asymptotically less storage than the size of the variable times the number of updates made to the variable [RS]. We are interested in the more general question of the complexity of computation using write-once memory. Though the questions that are addressed in this paper are not unrelated to previous work, the differences suggest a different model and approach.

In the write-once model of computation presented here, there are a small number of regular memory registers, each of which can store $\log n$ bits, where n is the problem size. The number of registers is given by some predetermined function of n, i.e., $f(n) = c$, $f(n) = \log n$, $f(n) = n^2$. The amount of write-once memory available is polynomial in the problem size. We assume that $O(\log n)$ consecutive bits can be read from or written to in one time step. This is a natural assumption, because it allows us to read polynomially bounded numbers in

constant time. In particular, this allows addressing to be performed in constant time. In addition, it is natural to assume that when executing a graph algorithm we can access a label of a node in constant time, or that in sorting elements, we can access the value of an element in constant time. The models in [Vi2] are different in that they assume that only a constant number of bits can be read from or written to in one time step. Their assumption is appropriate when looking at database problems, because there is no natural notion of problem size, i.e., address size and register size are considered to be a hardware-dependent constant. An additional difference in [Vi2] is that they do not allow the regular memory registers to be used as intermediate storage. In our model this would be equivalent to the assumption that an adversary may clear the regular memory registers between updates to the write-once memory machine. We call this version of our model the *register-restricted* version.

Techniques in [Vi2] give a method for converting any algorithm written for the conventional model of computation into an algorithm for the write-once model of computation which increase the running time by a factor of $O(\log n)$. (Their techniques are optimal on their model.) We show that the order of the running time need only be increased by a multiplicative factor of $O(\log n/\log\log n)$ on our model. There is an online problem in [Vi2] which requires $\Omega(n \log n/\log\log n)$ steps on the register-restricted write-once memory model, but only n steps on a regular memory model.

A slowdown of $O(\log n/\log\log n)$ is not required for all problems, and, in fact, standard algorithms for many well-known problems can be converted to algorithms that run on the write-once memory model of computation with no loss in speed, e.g., matrix multiplication, sorting, and shortest path computation. What are the properties which allow some algorithms to be converted with no loss in speed? One such property is obliviousness. An algorithm is *oblivious* if its read/write access pattern is the same for all inputs of the same length, and therefore depends only on the input size. We show that if there exists an oblivious algorithm for a problem running in time t on a regular memory machine, then there exists a nonuniform algorithm for the problem running in time $O(t)$ on the write-once memory model. The write-once algorithm can be found by preprocessing for a specific input size in $O(t)$ steps on a regular memory model or in $O(t \cdot \log t/\log\log n)$ steps on the write-once memory model. The preprocessing need only be done once for every input size, and therefore could be of practical use if the algorithm were to be run on many inputs of the same or similar size. In general, though, our feeling is that nonoblivious algorithms which involve complex data structures, especially ones in which elements may be pointed to from more than one other location, are likely to require some blowup in running time when simulated on write-once memory.

We show that any algorithm on regular memory using s space can be simulated with a multiplicative factor of $O(\log s)$ increase in the running time on write-once memory. When the value of s is small enough, this is an improvement over other techniques. We do not know if any multiplicative increase in running time is actually required for any problem on the general write-once memory model.

A *persistent* computation is one in which, at all times, the memory state of the computation at any previous point in time can be reconstructed. In [Vi2]

Vitter suggests the use of write-once memory in applications that require persistent data-structures. We show that any data-structure or computation on our write-once memory model can be made persistent with only an $O(\log \log n)$ multiplicative increase in time and an $O(\log n)$ multiplicative increase in space. Queries about the contents of memory location i at time j in the original (nonpersistent) computation can be answered in $O(\log \log n)$ time.

We consider a closely related but more restrictive model, where the write-once memory is replaced by memory in which a word may only be written to once. This model is interesting for two reasons: error detection can be done easily at the word level (i.e., parity check) and PROLOG is more accurately modeled. The proof of the previously described result also shows that any computation on the original write-once memory model can be simulated on the more restrictive model with only an $O(\log \log n)$ multiplicative increase in time and an $O(\log n)$ multiplicative increase in space.

The space requirements affect the cost-effectiveness of computing on write-once memory because it is not reusable. Rivest and Shamir investigate codes which allow making updates to a variable using asymptotically less storage than the size of the variable times the number of updates made to the variable [RS]. We examine the space requirements on write-once memory and consider the question of how to design algorithms so that the available regular memory can be used to conserve write-once memory space. On our model, many problems seem to require space proportional to the time required to solve them on a regular memory machine. It is easy to prove that on the register-restricted write-once memory model, the problem of maintaining variables through t on-line updates requires $\Omega(t)$ space. In fact, the results in [RS] show that maintaining a v-bit variable through t on-line updates on the register-restricted write-once memory model requires at least $t + v$ bits. We show that $\Omega(t)$ space is required on the general write-once memory model to maintain several variables. This implies that no general simulation technique exists which uses significantly less space than the number of steps being simulated. However, on the general write-once memory model, there are problems for which less space is needed because the regular memory can be used very effectively. In the pebble-sticker game there are pebbles *and* strickers. Pebbles correspond to the regular memory words, and stickers correspond to write-once memory words. We discuss the relationship between the number of stickers and pebbles required for trees and planar graphs.

Finally, the complexity class WO-PSPACE is defined to be the class of problems that can be solved in polynomial space on our model of a write-once memory machine with a constant number of regular registers. We show that it is equal to P.

The next section contains a description of the model. In Section 3 we discuss the time complexity of computing using write-once memory, and in Section 4 we discuss the space complexity. In Section 5 we define the complexity class WO-PSPACE and show that it is equal to P. We present our conclusions and open questions in Section 6.

2. The Model

Definition of Write-Once Memory. A bit of write-once memory may be set (changed from 0 to 1) but never reset (changed from 1 to 0). We assume that the hardware ignores commands that violate this restiction, and that initially all memory is set to 0.

Definition of Word-Write-Once Memory. A word in word-write-once memory may be written to only once. We assume that the hardware ignores commands that violate this restriction, and that initially all memory is set to 0.

Description of Model for Random Access Write-Once Memory Machine (WOMM). The model we use is a RAM with a small amount of regular memory and a polynomial amount of write-once memory. A memory word is defined to be $1 + d \cdot \log n$ bits where d is a constant and n is the problem size. (The first bit is used to decided whether to index into write-once memory or regular memory.) There are $C(n)$ regular memory words and n^d write-once memory words. $C(n)$ is a function which may vary. For example, we might choose either $C(n) = c$ or $C(n) = n^e$, depending on the amount of regular memory available. As long as the amount of regular memory is smaller than the amount of write-once memory, one memory word is large enough to address any word in the write-once or regular memory. We assume that all bits of write-once memory are initialized to 0 (except where the input data is stored). The space used by an algorithm is defined to be the number of memory words used by the algorithm. WOMM denotes the Write-Once Memory Model, word-WOMM denotes this same model with n^d words of word-write-once memory, and RMM denotes this same model with n^d words of regular memory instead of write-once memory.

We assume the normal constant time RAM operations on memory words such as copy, write, arithmetic operations, bit operations, jump, and two-way branches.

Disk Model. Current technology makes write-once memory available in disk form, thus it is appropriate to look at the effects of seek time on the computation time. We adopt the assumptions of [Vi2] with respect to seek time. In [Vi2] the seek time is considered to be the time required to access a single fixed length B-bit block different from the previously accessed block. The seek time is counted as a constant regardless of where the blocks are located in memory. There are various more complicated ways to model seek time, but since the effects of seek time are not well understood, even in the case of computing on regular memory, we adopt a relatively simple model as a first step. Clearly, whatever the assumptions, an upper bound on the number of steps is an upper bound on the number of seeks. The lower bound in [Vi2] yields a lower bound on the number of seeks on a model closely related to ours that has the same assumptions on seek time.

3. The Time Complexity of Computing on Write-Once Memory

3.1. Simulating Common Data Structures

We begin by showing how to maintain some simple common data structures in write-once memory. It is easy to see that on both the WOMM and word-WOMM we can perform n *queue* operations (enqueue and dequeue) in $O(n)$ time and n space using only two regular memory words to point to the head and the tail of the queue. We can also perform n *stack* operations (push and pop) in $O(n)$ time using only two regular memory registers on both the WOMM and word-WOMM. The write-once memory space required is $2 \cdot n$. The regular memory locations point to the top of the stack and to the first unused write-once memory location. When a push is done, the value of the push and the previous top of stack is written down at the first unused memory location. The regular memory locations are then updated accordingly.

In the next subsection we show that every data structure can be simulated with an extra $O(\log n/\log\log n)$ multiplicative time factor on the WOMM and an extra $O(\log n)$ multiplicative time factor on the word-WOMM. The question of what operations on various data structures can be simulated in less time is an important one. Unfortunately, it seems that very few of them can be simulated in the same order of time as on a regular machine.

One data structure which can be handled in on the WOMM and word-WOMM as quickly as it can be handled in regular memory is a binary tree in which the only pointers are from parents to children.

Claim. *We can perform n instert and delete operations on a binary tree of height h in $O(h)$ steps per operation, and $O(n)$ additional write-once memory space. Only two regular memory words are required.*

Proof. The tree operations can be done using the persistent search trees in [ST] in $O(n)$ space. We outline a simpler method that requires $O(n \cdot h)$ space. The pointer to the root of the tree is kept in a dedicated regular memory word. The internal nodes of the tree are kept in the write-once memory and contain pointers to the children of the node. Whenever node i is changed, all of the nodes along the path from the root to node i are copied to new locations in the write-once memory, updating the pointers appropriately and writing in the new value for node i. □

Surprisingly, the union-find data structure can also be implemented as quickly on a WOMM as on an RMM. To do this we first make the following observation:

Observation. We can implement a $\log n$ bit unary counter, using $O(n)$ additional space and $O(\log n)$ preprocessing time, such that increments and reads can be made in constant time.

The implementation uses a table that maps binary to unary and unary to binary. An entry in the table can be accessed in constant time.

Claim. *We can implement the set union algorithm with path compression and union by rank in $O(n\alpha(m, n))$ time on a WOMM, where n is the number of set-union operations and m is the number of elements in the set.*

Proof. We assume familiarity with the set union operations given in [Ta]. We say that an element is *involved* in a path compression step if the path compression step changes the pointer of the element. We say that an element is *affected* by a set union operation if, previous to the union operation, the element points to the root node, and the element is in the set with smaller rank (thus after the union operation, the element is no longer pointing to the root node). It is easy to see that in the time period between being affected by two union operations an element may be involved in at most one path compression step. Notice that when we implement the set union algorithm using union by rank, no element will be affected by more than $\log n + 1$ union operations because each time an element is affected, the size of the set that it is a member of must at least double. Thus, no element will be involved in more than $\log n + 1$ path compression steps. Since the pointer of the element changes only during a path compression step, or, in the case of the root element, during a union step, no element's pointer will be changed more than $O(\log n)$ times. Instead of allocating one pointer location to each element as is done on regular memory, $O(\log n)$ pointer locations are allocated to each element. Each time a pointer is changed, the new value of the pointer is written in the next consecutive location. In order to determine in constant time the location of the current value of the pointer, an $O(\log n)$ bit unary counter is kept, to which access (increment and read) can be made in constant time. □

It is interesting to note that in general it is not known how to maintain trees where the children point to the parent (such as the linking and cutting trees in [Ta]) as efficiently on write-once memory as on regular memory.

3.2. Simulating an RMM by a WOMM and word-WOMM

We discuss three upper bounds for simulating an RMM with a WOMM, which bound the simulation time by different quantities. None of the simulation upper bounds are better than the others in all cases, but the third is most general. All but the third also work on the word-WOMM. In each of the following we assume that the running times of the simulated algorithms are polynomial in n, where n is the size of the input.

Theorem 3.1. *If a program runs in time t and space s on an RMM, then it can be simulated on the WOMM/word-WOMM in $O(t \cdot \log s)$ steps and using a constant number of regular memory words and $O(t + s)$ write-once memory space.*

Proof. The values of the memory locations in the RMM are organized into a balanced binary search tree ordered by memory address. The pointer to the root of the tree is kept in a dedicated regular memory register. The values of the s memory locations are kept at the leaves of the tree. Whenever a memory location

in the simulated algorithm is changed, the value of the leaf associated with it is also changed. As explained in the previous section, each tree operation can be performed in $O(h)$ time where h is the height of the tree. The height of the tree is $\log s$. □

The following theorem follows from some elegant methods discussed in [Vi2] involving allocation trees. The proof presented here uses a different approach, but it is useful because it is simple and requires the storage of very few pointers.

Theorem 3.2. *Let A be an algorithm running on the RMM, whose running time is bounded by t, whose space requirement is bounded by s, and such that the number of updates to each location is bounded by b. Then A can be simulated in $O(t \log b)$ steps, a constant number of regular memory locations, and $O(t + s)$ write-once memory space on a WOMM/word-WOMM.*

Proof. In order to clarify the discussion, we refer to each memory location on the RMM as a variable. A k-block for a variable is a sequence of $2^k + 1$ consecutive locations, initially all 0. 2^k locations will be used to store 2^k updated versions of the variable, and the last location will be used as a pointer. The idea is to allocate initially to each variable a 0-block in write-once memory. When a k-block for a variable is filled up, a $(k + 1)$-block is allocated to the variable and the k-block is made to point to the $(k + 1)$-block. In order to find the value of the variable, a search is made for the most recently allocated block by following the address pointers until a block is reached that has no pointer filled in yet (this means the last one has been reached). Then a binary search is done on the block to find the last place in which a value was written. To change the value of the variable, the new value of the variable is written to the next location in the block. If at most b changes are made to a variable, then at most $\log b$ blocks are allocated to it. Therefore, following the addresses to the newest block takes at most $O(\log b)$ steps. The size of the block is $O(b)$ words, so the binary search to find the current value of the variable also takes at most $O(\log b)$ steps.

As stated, this scheme does not allow us to assign the value 0 to a variable. To fix this, another bit string can be used to represent the value 0. However, this still decreases the number of values that can be represented in a word by one. Another alternative is to keep a unary counter of 2^k bits with every block. The value of the counter indicates where the current value of the variable is stored in the block. This requires only $O(2^k/\log n)$ extra words. The binary search for the location of the current value is then done on the counter instead of the block itself. □

This last simulation is desirable, because it uses space efficiently. However, we can reduce the number of steps required if we are willing to use some extra space.

Theorem 3.3. *Let A be an algorithm whose running time is bounded by t, whose space requirements are bounded by s, and such that the number of updates to each*

location is bounded by b, on an RMM. Then A can be simulated on a WOMM using $O(t \cdot (\log b/\log\log n))$ *steps and* $O(t + s \cdot \log n \log b/\log\log n)$ *space.*

Corollary 3.1. *An algorithm that runs in* $O(t)$ *time on an RMM can be simulated in* $O((t \cdot \log t)/\log\log n)$ *steps on a WOMM.*

Proof. We describe a method which requires more space, but it is easy to see that it can be modified to run in the claimed space bound. We again refer to each memory location on the RMM as a variable. We show how to keep track of the current value of each variable in the program being simulated with $O(\log b/\log\log n)$ steps per access (read, write) to the variable, thus giving a method of simulating t steps of an algorithm in $O(t \cdot \log b)/\log\log n)$. The idea is to maintain for each variable a tree with $\log n$ degree at each node and b leaves. The jth leaf corresponds to the jth value that the variable takes on during the execution of the program. At any point in the computation, if the variable has changed fewer than j times, then the kth leaf is all 0's for all $k > j$. The internal nodes of the tree contain an address which is the address of the child that is on the path that leads to the leaf with the current value of the variable. It is not necessary for each internal node to contain pointers to all of its children, only the child which leads to the leaf with the current value of the variable. Therefore it is only necessary to store the children in $\log n$ addresses which are compatible in the sense that the address of the ith child can be changed into the address of the $(i + 1)$st child by only setting bits. This can be done by letting the first child be an address in which the last $\log n$ bits of the addresses are 0 and the ith address differs from the $(i - 1)$st address only in that the ith bit from the last is changed to 1.

By assumption, each variable can be changed at most b times during the execution of the algorithm. The depth of the tree, which is $O(\log b/\log\log n)$, is a bound on the time to access the corresponding variable. □

The proof of this theorem also shows that if B is the number of bits in a block, there is an $O(\log n/\log B)$ upper bound on the number of disk accesses required on write-once memory per disk access on regular memory. A lower bound in [Vi2] shows that this is tight on the register-restricted model.

Some algorithms can be solved on a WOMM in the same amount of time as on an RMM, even though only a small amount of information can be stored in the regular memory of a WOMM. In fact standard algorithms for determinants, matrix multiplication, and sorting work as quickly on a WOMM as they do on an RMM. We would like to characterize those properties of algorithms that allow this to be true. One such property, though by no means the only one, is the following:

Definition. An algorithm is *oblivious* if the read/write access pattern depends only on the size of the input, and not on its value.

Theorem 3.4. *If there is an oblivious algorithm for a problem that runs in time t on an RMM for input size n, then there exists a nonuniform (oblivious) algorithm*

which produces the same output and runs in time $O(t)$ on a WOMM/word-WOMM and uses $O(t)$ space. The preprocessing necessary to find the corresponding algorithm for a particular input size takes $O(t)$ time on an RMM, $O(t \cdot (\log t/\log\log n))$ time on a WOMM and $O(t \log t)$ time on a word-WOMM.

Proof. All of the operations defined in Section 2 can be decomposed into a constant number of read and/or write operations on at most a constant number of words, and arithmetic and logical operations on regular memory registers. Therefore we only need to show how to simulate t reads and writes in a total of $O(t)$ steps. A table is kept with an entry for each time step. When simulating a read at step i, the algorithm reads the ith entry in the table. We now show how to simulate a write of α to location j at step i. Suppose the next write after time i to location j is at time step i'. We write α to all entries in the table which correspond to the time steps in the original algorithm in which location j is read between steps i and i'. Because the algorithm is oblivious, the read/write accesses and therefore the information telling where to write in the table is the same for any input of size n. A total of t reads are made and, since the total number of writes in the simulation is bounded by the total number of reads, the nonuniform algorithm runs in time $O(t)$.

Finding the WOMM algorithm for a particular input size takes only $O(t \cdot (\log t/\log\log n))$ steps of preprocessing. It is done in two passes. In the first pass the algorithm is simulated on any input of size n. For each location, a list is kept of the time-steps in which reads and writes are made to that location. The lists can be constructed in $O(t \cdot (\log t/\log\log n))$ steps by keeping a counter for each variable indicating the number of times the variable has been accessed. t consecutive memory slots are alloted to contain descriptions of the accesses. The description consists of the type of access (read or write) and the time at which it occurs. The counter points to the next free slot in which to write the description of the next access to that variable.

In the second pass a table is made that has an entry for each time step. Each entry points to a linked list containing the locations which the simulation must write to at each time step. This table can be made in $O(t)$ time by running through the information gathered in the first phase and filling it into the table. If no write is made at that time step, the list is empty. □

The simulation can be of practical use in cases when the same program is used many times on data sets of similar sizes.

There is a lower bound implied by a proof in [Vi2] on the simulation time for a problem on the register-restricted version of our model. The problem is that of maintaining a variable through n updates, such that, at any point in time, the value of the variable can be correctly determined. The problem can be solved trivially in n total steps on an RMM. The problem requires $\Omega(n \cdot \log n/\log\log n)$ steps on the register-restricted WOMM and $\Omega(n \cdot \log n)$ steps on the register-restricted word-WOMM. This proof does not apply to the general write-once memory models.

In light of Theorem 3.4, showing that a problem in P requires asymptotically

more time on a WOMM than an RMM is a hard task: it would imply the problem cannot be solved by a linear-sized circuit, a major open problem in computational complexity. This is so because a circuit is an oblivious algorithm. (Actually, if only a uniform separation exists, then it shows that there are no (logspace, linear time)-uniform linear-size circuits, which is open as well).

Short of a major breakthrough in computational complexity, this gives hope only for showing lower bounds on on-line simulations of an RMM by a WOMM. By on-line simulations, we mean a simulation that keeps track of the value of each memory cell of the RMM. Our success in this task has not been better. However, we can identify a problem that is "complete" for the on-line simulation problem. The problem is the counter-maintenance problem: there are n counters initialized to 0, and we are given a series of t requests to either increment counter i or to report its value. The best-known algorithm on a WOMM requires $O(t \cdot \log t/\log\log n)$ steps. On the RMM, this problem can be done in $O(t)$ time, regardless of the number of times a counter is incremented. This problem is complete in the sense that if this problem can be solved in $O(u)$ total time on a WOMM, then any program requiring t steps and polynomial space can be simulated in $O(u)$ total time.

3.3. Making a Computation Persistent

Definition. A computation is called *persistent* if at any point in the computation, the state of the memory at any previous time of the computation can be reconstructed.

Theorem 3.5. *Any computation on a WOMM requiring t steps and s space can be made into a persistent computation running in $O(t \log\log n)$ steps and $t \cdot C(n) + s \log n$ space (where $C(n)$ is the number of regular memory words). Determining the contents of location i at time j can be done in $O(\log\log n)$ steps.*

Proof. For general $C(n)$, the state of the regular memory words at each point in time is kept in priority queues which allow accesses and predecessor computations in $O(\log\log n)$ time [EKZ]. There will be one priority queue for each regular memory word in the simulated computation. If the regular memory word in the simulated computation is updated to i at time j, $j \cdot (n+1) + i$ will be inserted into the priority queue associated with that regular memory word. The value of regular memory word i at time j can be retrieved by asking for the predecessor of $(j+1) \cdot (n+1)$ and taking the value to be the value of the predecessor mod n. $O(C(n))$ of these data structures can be implemented using $O(C(n))$ registers of regular memory and the write-once memory.

In order to keep the state of the write-once memory words at each point in time, we first observe that a write-once memory word can only be changed $1 + d \log n$ times because each change sets at least one bit and each bit can be set at most once. Thus we can simulate each location i in the original computation using $2(1 + d \log n) + 1$ consecutive locations in the persistent computation (where $1 + d \log n$ is the number of bits in a memory word) in the following way: Initially

blank, the first $2(1 + d \log n)$ consecutive locations will contain a "history" of location i in the original computation. The history will be of the form of $(1 + d \log n)$ ordered pairs (time stamp, value). If the jth change of location i in the original computation was made at time u by writing v, then the jth ordered pair will be (u, v). When location i is changed in the original computation, a new ordered pair can be inserted in the next consecutive blank locations. Letting the last location act as a unary counter, the next blank location can be found in constant time. The value of location i at time u in the original computation can be found by binary search on the time stamps. □

This proof also shows that any computation on the WOMM can be simulated on the word-WOMM with only an $O(\log \log n)$ multiplicative increase in time and an $O(\log n)$ multiplicative increase in space.

4. The Space Complexity of Computing on Write-Once Memory

In this section we investigate how efficiently space can be used in write-once memory. In the situation where we would like to keep track of a variable through several changes without using any regular memory, [RS] shows how to conserve the number of write-once memory bits required. However, if t changes are to be made to the variable, t write-once bits are necessary. Our emphasis is different because a WOMM has a certain amount of regular memory which can be used. We are interested in modifying algorithms in order to use the regular memory to conserve space.

It seems that many algorithms on the WOMM require space proportional to the running time. The following theorem shows that space proportional to the number of simulated steps is required for any general simulation of a program originally running on an RMM.

Theorem 4.1. *If $k \geq c \cdot d + 1$, where c is the number of $d \cdot \log n$ bit regular memory words available, then maintaining k variables given n on-line updates requires $\Omega(n)$ bits of write-once memory. Each update changes exactly one variable to any value in $[1, \ldots, n]$.*

Proof. We prove the theorem for the case $k = cd + 1$. The general theorem follows trivially from this. We show that an adversary can force a write to a write-once memory location after every k steps. Define a *register configuration* to be a snapshot of the registers, and a *memory configuration* to be a snapshot of the registers and the write-once memory. Note that there are only n^{cd} possible register configurations. Define a state of the k variables to be a k-tuple $(x_1, \ldots, x_k)$ where $x_i \in [1, \ldots, n]$ is the current value of the ith variable. Since the algorithm is maintaining the variables, it must be able to find out the value of each variable at all times. As the information about the variables can be assumed to be contained solely in the memory, we know that no two states of the k variables can have the same memory configuration. (On the other hand, it is possible that more than one memory configuration could indicate the same state since the memory configura-

tion may be dependent on the order of the updates.) Starting from any state, any one of at least n^k different states can be reached after k steps. Since $n^k > n^{c \cdot d}$, there is at least one pair of states that have the same register configuration. Since the memory configurations must be different for different states, there must have been a write to the write-once memory for at least one of these two states. Therefore, there is a way of updating the variables in order to force a write to the write-once memory every $k = cd + 1$ steps. □

A similar proof shows that for large enough k the lower bound holds even when the variables are counters that may only be incremented by one at each update.

Theorem 4.2. *Maintaining $n^{1-\varepsilon}$ counters given n on-line updates requires $\Omega(n)$ write-once bits.*

There are, however, algorithms that require significantly less write-once space than their running time on an RAM. For example, the "high school" method for Gaussian elimination takes time $O(n^3)$. At each phase, the new matrix that is calculated after doing the row operations is written down. This method takes $O(n^3)$ space. However, a factor of n can be saved in the space without affecting the running time by saving the row operations rather than the current values of each row. Therefore, the space required is only $O(n^2)$. In order to study space requirements further, we define a variant of the pebbling game, called the pebble-sticker game.

Pebbling graphs is a common tool used in examining the space requirements and the time–space tradeoffs in oblivious computation (see [P] for a survey). The idea is to model an algorithm by a directed acyclic graph. The nodes with zero indegree correspond to the inputs and the nodes with zero outdegree correspond to the outputs. The interior nodes correspond to operations. There is a directed edge from a node u to a node v if the output of node u is an operand for v. We say that u is a *direct predecessor* of v. The object of the pebbling game is to cover each vertex of a graph with a pebble, subject to the condition that before a pebble can be placed on a vertex v, all direct predecessors of v must be covered by pebbles. A pebble can be removed from a vertex at any time. The number of pebbles required to pebble a graph represents the space requirements of the computation and the number of steps corresponds to the computation time. The problem is to find the minimum number of pebbles needed to cover the graph or to find tradeoffs between the number of pebbles and the number of steps.

The analogous problem with write-once memory uses stickers in addition to pebbles. A sticker is different from a pebble in that once a sticker has been placed on a node, it cannot be removed. As placing pebbles correspond to writes to the regular memory, placing stickers correspond to writes to write-once memory. The problem is to find the minimum number of stickers required to cover a graph, given only a limited number of pebbles. Bounded degree planar graphs can be covered with $O(\sqrt{n})$ pebbles [LT], and there exist bounded degree planar graphs which require $\Omega(\sqrt{n})$ pebbles [M]. Bounded degree trees can be covered with

$O(\log n)$ pebbles, and balanced binary trees require $\log n + 1$ pebbles [PH]. The following four theorems show tight bounds on the number of stickers required to pebble directed acyclic planar graphs and trees, given a limited number of pebbles.

Theorem 4.3. *Directed acyclic planar graphs of indegree less than $p/(4 \log p)$ can be covered with $O(n/p)$ stickers, where n is the size of the graph and p is the number of pebbles.*

Proof. The method in [LT] for pebbling planar graphs uses the fact that, for any planar graph $G = (V, E)$, the vertices of G can be partitioned into three sets, A, B, C, where $n/3 \leq |A| \leq 2n/3$ and $|C| \leq 2\sqrt{2 \cdot n}$, such that there is no edge in G between a vertex of A and a vertex of B. We use this fact to define a tree structure, $T(G)$, on the graph such that each node in the tree is associated with a "small" subset of the nodes in the graph G. We say that a node in the tree *contains* the set of nodes in G that it is associated with. For a node v in the graph, let *Node* (v) be the node in the tree $T(G)$ where v is contained. The sets contained in each node of the tree form a partition of the nodes in G. Let A, B, and C be the compoinents of G as defined above (note that A, B, and C form a partition of V). Let G_A be the subgraph induced by the vertex set A and G_B be the subgraph induced by the vertex set B. Define $T(G)$ recursively as follows: if $|V| < p^2/4$, then $T(G)$ is just a one-node tree that contains the set V. Otherwise, the root of tree $T(G)$ contains the nodes in component C. The right subtree of $T(G)$ is $T(G_A)$ and the left subtree is $T(G_B)$.

Observation. If two nodes, u and v, are adjacent in G, then either *Node* (u) = *Node* (v), *Node* (u) is an ancestor of *Node* (v), or *Node* (v) is an ancestor of *Node* (u).

We use stickers to cover all graph nodes v such that *Node* (v) is an interior node in the tree $T(G)$. We call these nodes *sticker nodes*. All other nodes are called *pebble nodes* and are covered by pebbles. We cover the graph in topological order. It follows directly from the results of [LT] that if the degree of the graph is bounded by $p/(4 \cdot \log p)$, then the subgraph induced by the nodes at each leaf in $T(G)$ can be covered using only $p/2$ pebbles and without using any stickers. Therefore, if a node v is a pebble node and all of the sticker nodes that precede v in the topological ordering are already covered, then any node having an unblocked path (a path with no sticker or pebble on it) to v is contained in *Node* (v) and thus v can be covered using $p/2$ pebbles. To cover a sticker node v in the graph, we assume that all sticker nodes that precede v in the topological ordering have been covered. If u is an uncovered direct predecessor of v, then u is a pebble node such that all of u's predecessors that are sticker nodes have been covered. We cover u using $p/2$ pebbles and leave the pebble on u. When all of the direct predecessors of v have been covered, then v can also be covered.

Let $S(n, p)$ be the number of stickers required to cover a planar graph on n vertices, using only $p/2$ pebbles. We then have

$$S(n, p) \leq 2\sqrt{2 \cdot n} + S(\alpha n, p) + S((1 - \alpha)m, p),$$

where

$$\tfrac{1}{3} \leq \alpha \leq \tfrac{2}{3}$$

and

$$S\left(\frac{p^2}{4}, p\right) = 0,$$

which gives $S(n, p) = O(n/p)$. □

Theorem 4.4. *For all $0 < p < \sqrt{n}/3$, there is a family of bounded degree planar graphs that requires $\Omega(n/p)$ stickers given at most p pebbles.*

Proof. A mountain range is a directed, acyclic, planar graph with vertex set $\{1, \ldots, n\}$. The edge set is defined in terms of an auxiliary height function h from the vertex set into the nonnegative integers satisfying $h(1) = h(n) = 0$ and $|h(i + 1) - h(i)| = 1$. There is an edge from i to j if and only if $j = i + 1$ or $j = \min\{k > i | h(k) = h(i)\}$. We define a peak of a mountain range to be a subgraph induced by a sequence of nodes, $[i, \ldots, j]$, where $h(i) = h(j) = 0$ and $h(k) \neq 0$ for $k \in [i + 1, \ldots, j - 1]$.

At least r pebbles are required to pebble a mountain range that has r peaks of height r. The size of the smallest such graph is $n = 2r^2 + 1$. The case where $r = 3$ is shown in Figure 1. (See [M].)

Now examine the mountain range of size n with $n/3p$ peaks each of height $3p/2$. Divide the peaks into sections of $3p/2$ consecutive peaks. This gives $2n/9p^2$ sections. Since $p < \sqrt{n}/3$, there is at least one section of peaks. Each section requires $3p/2$ pebbles to be covered [M]. If only p pebbles are available, then at least $p/2$ stickers are required for each section because the number of pebbles plus the number of stickers must be at least $3p/2$ for each section. Since stickers cannot be reused, and there are $2n/9p^2$ sections, $n/9p$ stickers are required to cover the whole graph. □

Theorem 4.5. *Any binary tree can be covered using $O(n/2^p)$ stickers where n is the number of nodes in the tree and p is the number of available pebbles.*

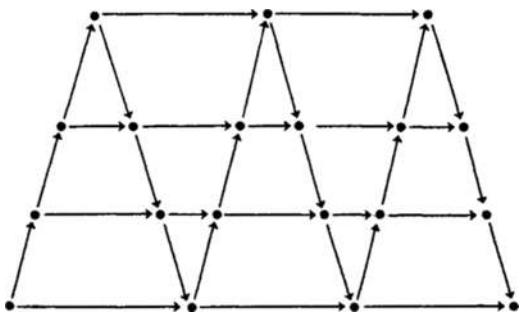

Fig. 1

Proof. We show that there are $O(n/2^p)$ nodes whose removal results in components which are all of size smaller than 2^p nodes. Any tree of size 2^p nodes can be pebbled with p pebbles. The entire graph can be covered by using stickers to cover $O(n/2^p)$ nodes that partion the tree into subtrees of size at most 2^p and using p pebbles to cover each of the nodes in the subtree. Such a partition can be shown to exist as we now describe. Every binary degree tree has a node, v, such that

$$\tfrac{1}{3}n \leq T_v \leq \tfrac{2}{3}n,$$

where T_v is the number of nodes in the subtree rooted at v [B]. If this node is removed, then there are two subgraphs, each with fewer than $2 \cdot n/3$ nodes. Let $T(n, p)$ be the number of nodes that must be removed from any binary tree on n nodes to obtain components that are all smaller than 2^p nodes:

$$T(n, p) \leq T(\alpha \cdot n, p) + T((1 - \alpha)n, p) + 1,$$

where

$$\tfrac{1}{3} \leq \alpha \leq \tfrac{2}{3}$$

and

$$T(2^p, p) = 0.$$

Hence,

$$T(n, p) = O\left(\frac{n}{2^p}\right).$$ □

The next theorem shows that this is the best possible.

Theorem 4.6. *$\Omega(n/2^{p+1})$ stickers are required in order to cover balanced binary trees with edges directed toward the root, where n is the number of nodes in the tree and p is the number of available pebbles.*

Proof. Let T be the complete binary tree of height h with $n = 2^{k+1}$ nodes. Let v be a node in such a tree and let $P(v)$ be the number of pebbles required to cover v. It is known that $h + 1$ pebbles are required to cover T [PH]. Now consider the nodes in the tree that are at distance p from the leaves. There are 2^{h-p} such nodes. A subtree rooted at one of these nodes is balanced and has 2^{p+1} nodes. Thus, it takes $p + 1$ pebbles to pebble one of these subtrees. If we only have p pebbles available, we must use at least one sticker to cover the subtree. Since there are 2^{h-p} such subtrees, we must use at least $2^{h-p} = n/2^{p+1}$ stickers to cover the entire binary tree. □

5. Relationships Between Write-Once Complexity Classes and Other Complexity Classes

Definition. WO-PSPACE is the class of problems that can be solved in polynomial space on a WOMM where the number of regular memory registers available is $C(n) = c$.

Theorem 5.1. WO-PSPACE = P.

Proof. As a result of our simulation upper bounds, it is clear that anything that is in P is also in WO-PSPACE. We now show that a problem in WO-PSPACE is in P. First notice that only a polynomial number of writes to the write-once memory can be made, because each write sets at least one bit. This implies that the write-once memory can only be in a polynomial number of configurations throughout the course of the computation. Since there are only $c \cdot d \cdot \log n$ bits of regular memory, the regular memory can only be in one of $n^{c \cdot d}$ configurations. Therefore the number of memory configurations is bounded by a polynomial in n. Each operation depends on the current instruction (of which there are a fixed number) and on the memory configuration. Since the computation terminates, no two time steps have the same memory configuration and current instruction. Therefore there can only be a polynomial number of operations. □

6. Conclusions and Further Questions

In this paper we have introduced a model for computation with write-once memory. We found that several algorithms can be easily converted to run as quickly on this model as on an RAM with regular memory, but that others seem to require some slowdown. We make an attempt to characterize the reasons for this difference.

All of our simulation time upper bounds use only a constant number of regular memory words. What better time bounds can be found for simulations on the WOMM when the number of regular memory words is more than a constant?

As noted before, it would be of interest to find problems for which the time or space complexity is provably greater on the WOMM than it is on the RMM. On the other hand, as is the case in parallel complexity classes, there are many problems for which it should be possible to find upper bounds on time and space which are better than those given by simulation results. For example, can maximum flow problems be solved as quickly on a WOMM?

The branching program model was used to investigate time–space tradeoffs of general computation. There is a natural analogue of this model in write-once memory. Is there a stronger time–space tradeoff lower bound for sorting on this model than the $\Omega(n^2)$ lower bound in [BC] for branching programs on regular memory? Given $o(n)$ write-once memory words for free, can it be shown that there is an $\Omega(n^2)$ lower bound on the time–regular-memory-space tradeoff? On the other hand, there are a few known $O(n \log n)$-time randomized algorithms for sorting which use $O(n)$ words on a WOMM, but are there any such deterministic algorithms?

Can algorithms for maintaining persistent data-structures be found which run on a WOMM with comparable bounds on time and space as those achieved on an RMM in [DSST]?

We could consider extensions of this model to models of parallel computation. Similar simulation results could again be used to give algorithms on PRAMs with

write-once memory with slightly worse running times than those on PRAMs with regular memory, but the above questions for sequential complexity are still relevant with respect to parallel complexity.

Though this model is incomparable with the Hierarchical Memory Model with Block Transfer model defined in [ACS], it seems that many of the same problems that can be done with little slowdown on that model can also be done with little slowdown on a WOMM. It would be interesting to find out if this is because of the oblivious nature of the algorithms exhibited in [ACS], or if there is a deeper reason for this to be the case.

Finally, the model could be extended to incorporate more sophisticated ways of modeling seek time. For example, it would be more accurate to distinguish between consecutive and nonconsecutive reads when charging for a step. If no seek is required for a consecutive read, how is the number of seeks required affected?

Acknowledgments

We than Manuel Blum and Raimund Seidel for suggesting this area of research, for many helpful conversations, and for Raimund's suggestion of the use of persistent data structures to improve the space bound on the claim in Section 3.1. We thank Mike Fredman for pointing us to $O(\log \log n)$ priority queues. We also thank Mike Luby for his careful reading and comments on this paper, and Russell Impagliazzo, Ron Rivest, Steven Rudich, and Umesh Vazirani for several interesting discussions.

References

[ACS] Aggarwal, A., Chandra, A., Snir, M., Hierarchical Memory with Block Transfer, *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, Los Angeles, CA, October 1987, pp. 204–216.

[ANP] Arvind, Nikhil, R. S., Pingali, K. K., I-structures: Data Structures for Parallel Computing, *ACM Transactions on Programming Languages and Systems*, Vol. 11, No. 4, October 1989, pp. 598–632.

[BC] Borodin, A., Cook, S., A Time-Space Tradeoff for Sorting on a General Sequential Model of Computation, *SIAM Journal on Computing*, Vol. 11, No. 2, May 1982, pp. 287–297.

[B] Brent, R. P., The Parallel Evaluation of General Arithmetic Expressions, *Journal of the Association for Computing Machinery*, Vol. 21, 1974, pp. 201–208.

[DMMU] Dolev, D., Maier, D., Mairson, H., Ullman, J., Correcting Faults in a Write-Once Memory, *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, Washington, DC, May 1984, pp. 225–229.

[DSST] Driscoll, J., Sarnak, N., Sleator, D., Tarjan, R., Making Data Structures Persistent, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, Berkeley, CA, May 1986, pp. 109–121.

[EKZ] Emde Boas, P. v., Kass, R., Zijlstra, E., Design and Implementation of an Efficient Priority Queue, *Mathematical Systems Theory*, Vol. 10, 1977, pp. 99–127.

[LT] Lipton, R., Tarjan, R., Applications of a Planar Separator Theorem, *SIAM Journal on Computing*, Vol. 9, No. 3, August 1980, pp. 615–626.

[M] Mehlhorn, K., Pebbling Mountain Ranges and Its Application to DCFL-Recognition, 1979.

[PH] Paterson, M. S., Hewitt, C. E., Comparative Schematology, *Proceedings of the MAC Conference on Concurrent Systems and Parallel Computation*, 1970, pp. 119–127.

[P] Pippenger, N., Pebbling, *Proceedings of the 5th IBM Symposium on Mathematical Foundations of Computer Science: Computational Complexity*, May 19, 1980.

[PMN] Ponder, C., McGerr, P., Ng, A., Are Applicative Languages Inefficient? *SIGPLAN Notices*, Vol. 23, No. 6.

[RS] Rivest, R. L., Shamir, A., How To Reuse a "Write-Once" Memory, *Information and Control*, Vol. 55, Nos. 1–3, 1982, pp. 1–19.

[ST] Sarnak, N., Tarjan, R. E., Planar Point Location Using Persistent Search Trees, *Communications of the ACM*, July 1986, Vol. 29, No. 7, pp. 669–679.

[Ta] Tarjan, R. E., *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, Philadelphia, 1983.

[To] Touati, H., Personal communication.

[Va] Valiant, L. G., Graph-Theoretic Arguments in Low-Level Complexity, *Theoretical Computer Science*, 1977.

[Vi1] Vitter, J. S., Computational Complexity of an Optical Disk Interface, *Proceedings of the 11th Annual International Colloquium on Automata, Languages, and Programming (ICALP)*, Antwerp, July 1984, pp. 490–502.

[Vi2] Vitter, J. S., An Efficient I/O Interface for Optical Disks, *ACM Transactions on Database Systems*, Vol. 10, No. 2, June 1985, pp. 129–162.

Received May 2, 1991.