CLOCKED ADVERSARIES FOR HASHING

Richard J. Lipton
Jeffrey F. Naughton

CS-TR-203-89

February 1989

Clocked Adversaries for Hashing*

Richard J. Lipton and Jeffrey F. Naughton
Computer Science Department
Princeton University

February 1, 1989

Abstract

A "clocked adversary" is a program that can time its operations and base its behavior on the results of those timings. While it is well known that hashing performs poorly in the worst case, recent results have proven that for reference string programs, the probability of falling into a bad case can be driven arbitrarily low. We show that this is not true for clocked adversaries. This emphasizes the limits on the applicability of theorems on the behavior of hashing schemes on reference string programs, and raises a novel set of problems dealing with optimality of and vulnerability to clocked adversaries.

1 Introduction

Traditionally, the performance of hashing schemes has been measured by modeling programs as reference strings. This approach reduces a program to a sequence of memory references, and then measures the performance of the hashing scheme when presented with that string of references. However, in actual systems, programs are not just reference strings. Programs have access to clocks, and can dynamically change their behavior based on the performance of previous operations. To model this larger class of programs, we define *clocked adversaries.* This is a simple concept; a clocked adversary is just a program that can time its operations, and base its behavior on the results of these timings.

*Work supported by DARPA and ONR contracts NOOO14-85-C-0456 and N00014-85-K-0465, and by NSF Cooperative Agreement DCR-8420948

When one considers clocked adversaries, the expected performance of hashing schemes must be re-examined. Intuitively, a clocked adversary can detect when two keys collide, and use this knowledge to its advantage. In this paper we consider the implications of clocked adversaries on two hashing applications. The first is universal hashing; the second is PRAM simulation.

In *universal hashing*, a hash function is chosen at random from a family of universal$_2$ hash functions. Carter and Wegman [CW79] have shown that this technique provides $O(n)$ expected time for reference string programs of $O(n)$ associative memory operations. In Section 3, for each of the families of hash functions H_1, H_2, and H_3 presented in [CW79], we give clocked adversary programs that make $O(n)$ associative memory operations, yet take expected time $\Omega(n^2/\log n)$, $\Omega(n^2)$, and $\Omega(n^2)$ respectively. Whether there are universal$_2$ families of hash functions that are not vulnerable to such an attack is an interesting open question.

The second line of research to which we apply clocked adversaries is the simulation of PRAMs on message passing, bounded degree architectures. There is a growing body of literature proposing that this be done by hashing the PRAM memory locations to the various memories in the actual machine. These papers show that for reference string programs, the expected time degradation in this simulation is at worst logarithmic. In Section 4 we show that, for any function $T(n) \geq n(\log n)^2$, there is a clocked adversary PRAM program that runs in time $O(T(n))$ on a PRAM, yet takes expected time $\Omega(nT(n))$ under the hash-based simulations.

Of course, it has always been known that in the worst case, hashing performs abysmally. However, papers such as [CW79,KU86,Ran87] have essentially shown that the probability of the worst case occuring can be made arbitrarily small. Such results begin to carry the weight of absolute performance guarantees, since the probability of poor behavior apparently can be driven far below the probability of machine failures. The work presented in this paper does not dispute these results; however, it emphasizes that they hold only for a subset of the programs that can be written on actual machines.

2 Clocked Adversaries

As mentioned in the introduction, a clocked adversary is a program that has access to a clock. The critical capability of clocked adversaries is given by the following lemma.

Lemma 2.1 *Let s_1 and s_2 be two possible states of a data structure D, and let o be some operation on D such that*

- *D does not change state in response to operation o, and*
- *operation o takes time t_1 in situation s_1 and time t_2 in situation s_2, and*
- *$t_1 - t_2 \geq \delta$.*

Then a clocked adversary with access to a clock that is accurate to within ϵ can detect the difference between s_1 and s_2 with $O(\epsilon/\delta)$ repetitions of operation o.

Proof: Because the data structure D does not change state in response to o, the adversary can repeat o as often as required. Repeating o a total of m times in situation s_1 will take mt_1 time, whereas in situation s_2 it will take mt_2. Because $t_1 - t_2 \geq \delta$, we have $mt_1 - mt_2 \geq m\delta$.

For the adversary to detect the difference between the two, we require that $m\delta \geq \epsilon$, or that $m \geq \epsilon/\delta$, which completes the proof. □

As a concrete example, consider a hash table T in which the hash function h hashes keys into the bins of T. For definiteness, we will assume that *chained-overflow* collision resolution is used[1], although the adversaries presented in this paper can be adapted for other static collision resolution strategies (e.g., open addressing.)

Suppose that we wish to determine whether or not two keys k_1 and k_2 collide under h. Lemma 2.1 applies with s_1 the situation where k_1 and k_2 collide, s_2 the situation where they do not, and o the operation *fetch(k)*, for a key k. Here δ is the difference between the time to fetch a key at the head of a bin's linked list and the time to fetch a key at some other position in a bin's list. Letting t be the time to fetch a key inserted into a previously empty table (this ensures that t is the time to fetch an element at the head of the list in a bin) we construct the following adversary.

- Insert k_1 and k_2 into an empty table T.
- Let t' be the time it takes to repeat the operation *fetch(k_1)* ϵ/δ times.
- If $t' \geq t + \epsilon$, then report a collision; otherwise, let t'' be the time it takes to repeat the operation *fetch(k_2)* ϵ/δ times.

[1]Each bin b is the head of a linked list containing all the keys inserted into T that hash via h to b.

- If $t'' \geq t + \epsilon$, then report a collision; otherwise, report that k_1 and k_2 do not collide.

Real machines have widely varying δ's. For example, on an IBM PC, $\delta \approx 50$ milliseconds, whereas on a Cray-2 $\delta \approx 4$ nanoseconds. One implication of the Lemma 2.1 is that the resolution of the clock does not matter except that changing the resolution of the clock changes the constant factor in the running time of an adversary. In view of this fact, for the rest of this paper we will assume that the adversary's clock resolution is sufficiently high that $\epsilon < \delta$, so the difference between two situations of interest can be determined by timing a single operation.

As another example suppose we are given a set S of keys, and wish to determine if S contains a colliding pair of keys and, if so, wish to identify one such colliding pair. The following proposition demonstrates how a clocked adversary can do so in time $O(|S|)$.

Lemma 2.2 *Let T be a hash table, and let S be a set of keys. Then there is a clocked adversary that makes $O(|S|)$ insertions, deletions, and retrievals, correctly determines if S contains a pair of keys that collide in T, and, if so, returns a pair of colliding keys.*

Proof: Consider the following approach.

First, insert some key k from S into the empty table T, and time the retrieval of k. Let that time be t_0 — this is the time to retrieve a key that does not collide with any other key in the table. Next, insert the remaining keys of S into the table, and time the retrieval of every key in S. If no key in S has a retrieval time $t > t_0$, then there must be no collisions; otherwise, there is at least one collision.

Suppose that there is a collision. Then one of the keys participating in the collision, say k', must have a retrieval time greater than t_0. Delete all the elements of S from T except k'.

Next, for each key $k'' \neq k'$, insert k'', time the retrieval of k' and k'', then delete k''. Because k' collides with some key, there must be at least one k'' such that when k'' is inserted into T, the retrieval of k' or k'' is slowed. When such a k'' is detected, report that k' and k'' collide.

This adversary makes $O(|S|)$ insertions, deletions, and retrievals, and finds a colliding pair if one exists. □

We close this section with a lemma about generating pairs of colliding keys. Note that this lemma makes no assumptions about the hash function being used; the adversary used in the proof of this lemma will be useful as a subroutine in the next section.

Lemma 2.3 *Let n be the number of bins in a hash table T. Then, for any $\epsilon > 0$, there exists a clocked adversary that generates a pair of colliding keys with probability $p > 1 - \epsilon$. Furthermore, the adversary need only make $O(\sqrt{n})$ insertions, deletions, and retrievals.*

Proof: Consider generating $\sqrt{n}$ random keys. If the hash function h distributes the keys uniformly among the bins in the table T, then by a classical result from probability (the "birthday paradox"), with probability approximately $p \approx 1/2$ two of these keys will hash to the same bin. It is straightforward to show that if h distributes the keys nonuniformly, then with probability $q > p$ two of the keys will hash to the same bin.

This implies that if we iteratively apply the adversary from Proposition 2.2 k times on randomly generated sets of size $\sqrt{n}$, we will get a collision on at least one of the iterations with probability at least $1 - (1/2)^k$. Then for any ϵ, where $0 < \epsilon < 1$, by choosing $k > \log \epsilon$ we get a colliding pair of keys with probability $p > 1 - \epsilon$. □

3 Adversaries for Universal Hashing

In universal hashing, a hash function as chosen at random from a family of hash functions. Carter and Wegman [CW79] have shown that if the family of hash functions is universal$_2$, then the expected time for a sequence of n associative memory operations is $O(n)$.

Carter and Wegman also give three examples of universal$_2$ functions. In this section we give adversaries for each family. We assume that the adversary knows which family of hash functions the hash table uses, but does not know the specific function the table has chosen from within that family. Each of the adversaries shares the common theme of first using random probes to generate a small number of colliding pairs, then using these colliding pairs to generate a large set of keys that collide, and finally repeatedly retrieving a key in this set.[2]

Finding a small number of colliding pairs from random probes is straightforward, using the technique from the proof of Lemma 2.3. Generating a large set of colliding pairs from these colliding pairs is in general harder. In a sense is a cryptographic problem, because to solve it the adversary must learn something about the (unknown) hashing function the table uses. However, generating collisions is different from cracking codes in that it is not necessary for the adversary to completely determine the hash function. In fact, none of the adversaries below

[2]Recall that a hash table stores unique keys — an attempt to insert a key already in the table is an error. Because of this, the adversary must generate a large set of distinct keys.

actually discover the hash function; they just discover enough about the function to generate n keys that collide from $\log n$ pairs of colliding keys.

In the following we will assume that the keys are drawn from some domain A, while the bins are represented by the elements of another set B. Then each hash function is a mapping from A to B; two keys in A collide if they map to the same element of B. We will present the adversaries in order of difficulty: H_3, H_1, and finally H_2.

3.1 The Family H_3

In the family H_3, a generic function $h(x)$ is defined by the equation $h(x) = Mx + b$, where

1. M is a $\log|A| \times \log|B|$ boolean matrix, and
2. b is a $\log|B|$ bit boolean vector, and
3. multiplication is boolean and sum is exclusive or.

A specific function in H_3 is generated by choosing a matrix M and vector b.

Constructing a clocked adversary for the functions in H_3 is easy because the functions in H_3 are linear, as shown by the following lemma.

Lemma 3.1 *Let $h(x) = Mx \oplus b$ be a function from H_3, and let x and y be two keys in the domain of h. Then $h(x \oplus y) = h(x) \oplus h(y) \oplus b$.*

Proof: The proposition follows immediately from the associativity and commutativity of $\oplus$, and the fact that $\cdot$ distributes over $\oplus$. □

Note that if $h(x) = h(y)$, then the preceding proposition implies that $h(x \oplus y) = b$. This is the key idea in the following theorem.

Theorem 3.1 *Let T be a hash table using a hash function chosen at random from H_3. Then there exists a clocked adversary that makes $O(n)$ insertions, deletions, and retrievals and runs in time $\Omega(n^2)$.*

Proof: Consider the following adversary:

1. Find $\log n$ pairs of colliding keys by repeatedly applying the adversary from Lemma 2.3. Let these colliding pairs of keys be (x_i, y_i), for $1 \leq i \leq \log n$.

2. Let I be a subset of the interval $1 \leq i \leq \log n$. Construct the n possible sums $\sum_{i \in I}(x_i \oplus y_i)$.

3. Insert these sums into the table.

4. Time the retrieval of each of the sums. Let s be the sum with the maximum retrieval time.

5. Perform the operation "retrieve s" n times.

Again there are $O(n)$ sums of the form $\sum_{i \in I}(x_i \oplus y_i)$. By a straightforward application of Lemma 3.1, there are only two bins into which these sums will hash: b, if the cardinality of I is odd; 0 otherwise. This in turn implies that the bin containing the most keys must contain at least $n/2$ keys. The retrieval time for at least one of the keys in that bin must be proportional to n. Retrieving this key n times gives a running time that is $\Omega(n^2)$. □

3.2 The Family H_1

The family H_1 is defined by $h(x) = (\alpha x + \beta \bmod p) \bmod n$. Here, $n = |B|$, α and β are random integers, and p is some prime. The family of functions is generated by using multiple values of α, β, and p, generating one function for each triple (α, β, p). Our adversary for H_1 is more complicated than our adversary for H_3, because the functions in H_1 are nonlinear.

Our clocked adversary for this family of functions makes use of the following lemma.

Lemma 3.2 *Suppose that for* $1 \leq i \leq k$, $h(x_i) = h(y_i)$. *Then there are* $\Omega(2^k/k)$ *sets* $I \subseteq \{1, \ldots, k\}$ *such that*

$$h(I) = h\left(\sum_{i \in I}(x_i - y_i)\right)$$

all have the same value.

Proof: First, note that by definition of mod,

$$h(x) = \left(\alpha x + \beta - p\left\lfloor \frac{(\alpha x + \beta)}{p} \right\rfloor\right) \bmod n$$

Next, let $I \subseteq \{1, \ldots, k\}$ and define $l(I)$ by

$$\left\lfloor \sum_{i \in I}\left(\frac{\alpha(x_i - y_i) + \beta}{p}\right) \right\rfloor = \sum_{i \in I}\left\lfloor \frac{\alpha x_i + \beta}{p} \right\rfloor - \sum_{i \in I}\left\lfloor \frac{\alpha y_i + \beta}{p} \right\rfloor + l(I)$$

It is straightforward to show that $l(I) = O(k)$ for all sets I. Next, note that

$$\begin{aligned} h(I) &\equiv_n \sum_{i \in I} \left(\alpha(x_i - y_i) + \beta - p \left\lfloor \frac{\alpha(x_i - y_i) + \beta}{p} \right\rfloor \right) \\ &\equiv_n \left(\sum_{i \in I} (\alpha x_i + \beta) - p \sum_{i \in I} \left\lfloor \frac{\alpha x_i + \beta}{p} \right\rfloor \right) - \left(\sum_{i \in I} (\alpha y_i + \beta) - p \sum_{i \in I} \left\lfloor \frac{\alpha y_i + \beta}{p} \right\rfloor \right) + \sum_{i \in I} \beta - pl(I) \end{aligned}$$

We wish to bound the number of values the last line may take on, modulo n. First, note that the first term on the right side,

$$\left(\sum_{i \in I} (\alpha x_i + \beta) - p \sum_{i \in I} \left\lfloor \frac{\alpha x_i + \beta}{p} \right\rfloor \right)$$

is just $\sum_{i \in I} h(x_i)$. Similarly, the second term is just $\sum_{i \in I} h(y_i)$. So, replacing these terms, we have that the sum is just

$$\sum_{i \in I} h(x_i) - \sum_{i \in I} h(y_i) + \sum_{i \in I} \beta - pl(I) \bmod n$$

Because for each $h(x_i) = h(y_i)$ for each i, this is just $\sum_{i \in I} \beta - pl(I) \bmod n$. By definition of the mod function, this is

$$\sum_{i \in I} \beta - pl(I) - n \left\lfloor \frac{\sum_{i \in I} \beta - pl(I)}{n} \right\rfloor$$

Because β, n and p are independent of I, this can take on the same number of values as

$$-l(I) - n \lfloor -l(I)/n \rfloor$$

But, because $l(I) = O(k)$, this last quantity can take on at most $O(k)$ values.

To complete the proof, notice that there are 2^k subsets of the set $\{1, \ldots, k\}$. Since the quantity

$$h(I) = h \left(\sum_{i \in I} (x_i - y_i) \right)$$

can take on only $O(k)$ values, by the pigeon-hole principle, $\Omega(2^k/k)$ of these subsets must take on the same value. □

Lemma 3.2 allows us to state the following theorem about the existence of adversaries for the family of functions H_1.

Theorem 3.2 *Let T be a hash table using a hash function chosen at random from H_1. Then there exists a clocked adversary that makes $O(n)$ insertions, deletions, and retrievals and runs in time $\Omega(n^2/\log n)$.*

Proof: Consider the following adversary:

1. Find $\log n$ pairs of colliding keys by repeatedly applying the adversary of Lemma 2.3. Let these colliding pairs of keys be (x_i, y_i), for $1 \leq i \leq \log n$.

2. Form all n sums of the form $\sum_{i \in I}(x_i - y_i)$, where $I \subseteq \{1 \ldots \log n\}$.

3. Insert these sums into the hash table. By Lemma 3.2, some bin must contain $\Omega(n/\log n)$ keys.

4. Time the retrieval of all n sums just inserted into the table. Let s be the sum with a maximal retrieval time.

5. Perform the request "retrieve s" n times.

There are $\Omega(n)$ sums of the form $\sum_{i \in I}(x_i - y_i)$, and they can be generated in linear time. By Lemma 3.2, $\Omega(n/\log n)$ of these sums hash to the same bin.

If a bin in T contains $n/\log n$ keys, then the retrieval of at least one of these keys must take time proportional to $n/\log n$. Asking for this key n times gives a running time that is $\Omega(n^2/\log n)$. □.

3.3 The Family H_2

The family of hash functions H_2 is similar to the family H_3. However, before multiplying the key x by the boolean matrix A, the key x is transformed. The difficulty here is that the functions in H_2 are strongly nonlinear, requiring a more sophisticated attack than either H_1 or H_3.

A function from H_2 works as follows. For some integer d, consider the key x to be a number base d. (For example, if $d = 4$, then each pair of bits in the binary representation of x specifies one digit between 0 and 3.) Thus, setting $m = \log |B|$, and $r = \log d$, the key x is interpreted to be the number $d_1 d_2 \ldots d_{m/r}$. Next, re-represent x in unary by mapping $d_1 d_2 \ldots d_{m/r}$ to $0^{d_1}10^{d_2}10 \ldots 0^{d_{m/r}}10^{((r-1)m - d_1 - d_2 - \ldots - d_{m/r})}$. The resulting bit string is then multiplied by a $rm \times n$ boolean matrix M to give the bin corresponding to x.

For a key x, let $u(x) = x'$ be the result of interpreting x as a number base d and then re-expressing it in unary. The difficulty in constructing an adversary for H_2 is that $u(x)$ is nonlinear, so the hash functions in H_2, $h(x) = Mu(x) \oplus b$, are also nonlinear. To avoid this

difficulty we define a subset A' of the key domain A such that for any key x in A', $u(x)$ is linear. The set A' consists of all keys x such that, when x is interpreted as a number base d, the digits in x form a word in the regular set $\{12+21\}^{m/2r}$.

In the following, it is perhaps easiest to think of each member x of A' as being designated by a binary number $\hat{x}$ of $m/2r$ digits, where digit i of $\hat{x}$ is 1 if the ith pair of digits in x is 12, and digit i of $\hat{x}$ is 0 otherwise. We let D be the set of designators, and f be the map from a designator $\hat{x}$ to the corresponding key x. The key to the adversary for H_2 is the following lemma.

Lemma 3.3 *Let $\hat{x}$ and $\hat{y}$ be designators for keys x and y in A'. Then $u(f(\hat{x} \oplus \hat{y})) = u(f(\hat{x})) \oplus u(f(\hat{y})) \oplus c$, where $c = (0010)^k \ldots$*

Proof: Let $\hat{x} = x_1 x_2 \ldots x_k$ and $\hat{y} = y_1 y_2 \ldots y_k$. Then by definition of u and f, we have that

$$u(f(\hat{x})) = 0x_1(1 \oplus x_1)10x_2(1 \oplus x_2)1 \ldots$$

and

$$u(f(\hat{y})) = 0y_1(1 \oplus y_1)10y_2(1 \oplus y_2)1 \ldots$$

Again by definition of u and f, we have

$$\begin{aligned} u(f(\hat{x} \oplus \hat{y})) &= 0(x_1 \oplus y_1)(1 \oplus x_1 \oplus y_1) \ldots \\ &= 0(x_1 \oplus y_1)((1 \oplus x_1) \oplus (1 \oplus y_1) \oplus 1) \ldots \\ &= u(f(\hat{x})) \oplus u(f(\hat{y})) \oplus 00100010 \ldots \end{aligned}$$

□

In other words, for keys x in A', $u(x)$ is linear.

Lemma 3.4 *Let $\hat{x}$ and $\hat{y}$ be designators for two keys x and y in A', and let $h(x) = Mu(x) + b$ be a function from H_2. Then $h(f(\hat{x} \oplus \hat{y}) = h(f(\hat{x})) \oplus h(f(\hat{y})) \oplus b$.*

Proof: Similar to that of Lemma 3.1. Recall here that h is of the form $h(x) = Mu(x) \oplus b$, which can be written $h(x) = Mu(f(\hat{x})) \oplus b$. The key point is that f, u, are linear, thus so is their composition, which in turn implies that h is linear. A picture of the maps involved appears in Figure 1. In that figure, we let h' represent the function $h(y) = My \oplus b$. □

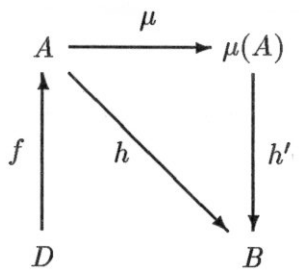

Figure 1: Mappings used in Lemma 3.4

Theorem 3.3 *Let T be a hash table using a hash function chosen at random from H_2, where that function interprets each key in A as a number base d. Furthermore, let $r = \log d$, and assume that $(\log |B|)/2r > 2 \log |A|$. Then there exists a clocked adversary that makes $O(n)$ insertions, deletions, and retrievals and runs in time $\Omega(n^2)$.*

Proof: Consider the following adversary.

1. Find $\log n$ pairs of colliding keys by a process similar to repeating the the adversary from Lemma 2.3. However, in this case, instead of generating the keys directly, generate designators, then work with the keys these designators designate. This ensures that we will only deal with keys in A'. Let the designators for the colliding pairs of keys be $(\hat{x}_i, \hat{y}_i)$, for $1 \leq i \leq \log n$.

2. Construct the n possible sums $\hat{s}_I = \sum_{i \in I} (\hat{x}_i \oplus \hat{y}_i)$, where $I \subseteq \{1 \ldots \sqrt{n}\}$

3. For each sum of designators $\hat{s}_I$, insert $f(\hat{s}_I)$ into the table.

4. Time the retrieval of $f(\hat{s}_I)$, for each I. Let $\hat{s}$ be the sum with the maximum retrieval time.

5. Perform the operation "retrieve $f(\hat{s})$ " n times.

This adversary works in for the same reason the adversary in Theorem 3.1 works — the hash function is linear (when restricted to A'.)

The technical restrictions on the sizes of A and B are necessary to ensure that there are enough keys in A'. □

4 Adversaries for PRAM Simulations

Much of the work in parallel algorithms has been devoted to algorithms for the PRAM model of computation. The most salient feature of this model is the assumption of a single, uniform, shared memory. This shared memory greatly simplifies the task of writing programs. However, it is difficult to provide such a memory in an actual machine, especially if the number of processors is large [AS88].

This has given rise to schemes for simulating PRAMs on multiprocessors without true shared memory. In the remainder of this section, the machine upon which the simulation of the PRAM is running will be called the *target machine.* We assume that the target machine consists of n processors, numbered 0 through $n-1$, each with an associated local memory. There is no global memory; however, the processors are connected by some interconnection network.

We will assume the target machine has the following properties:

- Each memory can only respond to one request (read or write) at a time, and
- Local memory references (processor i referencing a location in memory i) respond more quickly than remote references (processor i referencing a location in memory $j \neq i$.

Each assumption is well motivated by actual machines. Real memories can only respond to a small constant number of requests at a time. Also, a memory reference that need only use a local bus can be made to run much faster than a memory reference that requires a network message plus a local bus access. Any computer architect would take advantage of this fact to provide fast local memory.

If a PRAM program is to be simulated on a target machine, then the memory addresses of the PRAM program must be mapped to actual memory addresses in the memories of the target machine. The danger is that at some step, many processors may reference PRAM addresses within the same target machine memory. This creates a "hot spot," as n requests to the same target machine memory must take n target machine cycles to be serviced.

PRAM simulations attempt to eliminate hot spots in two ways. First, they use "combining networks" [PN85] that combine multiple requests for the same memory location as the requests pass through the network. This technique eliminates the possibility of hot spots due to multiple requests for the same PRAM address.

However, combining networks do not prohibit hot spots due to requests for distinct PRAM

addresses that map to the same target machine memory. This is where hashing comes in — the hope is that through a suitable hashing scheme, the probability of requests for a large number of distinct PRAM addresses within the same target machine memory is extremely low.

A number of papers have dealt with how to emulate a PRAM on a multiprocessor without shared memory [UW84,KU86,Ran87]. In Karlin and Upfal [KU86] and in Ranade [Ran87], it is proven that with the proper choice of the hashing, routing, and combining strategies, the target machine can simulate a single step of a PRAM in $O(\log n)$ with high probability. As we will show in this section, these proofs depend on the fact that in the PRAM model, programs do not have access to clocks. If we allow the class of simulated programs to contain clocked adversaries, then we can give a program that runs in time $O(T(n))$ on the PRAM, but in time $\Omega(nT(n))$ (rather than $O(T(n)\log n)$ on the target machine.)

Before stating our results about such adversaries, we note that in any target machine with more than one processor, every program is potentially a clocked adversary (not just those with explicit access to a clock.) This is made more precise by the following proposition.

Proposition 4.1 *Let M be a target machine with $n > 1$ processors, and let M' be an identical target machine except that one processor has been replaced with a clock. Then for every clocked adversary P' running on M', there is an equivalent adversary P that runs on M.*

Proof: Clearly, any segment of P' that makes no reference to a clock can be run on the $n-1$ processors of P, so we just need to show how P can simulate a clock. This can trivially be accomplished by dedicating one processor of P to continually incrementing a register r. Then calls to "get time" on P' are replaced by the operation "read r" in P. □

Note that in an uninteresting technical sense, PRAM programs cannot be clocked adversaries, because there is no notion of time (other than computational steps) in the PRAM model. Our intent is that the clocked adversary PRAM program is a program written to run on the PRAM simulator. From this perspective, the results of this section can be interpreted as emphasizing that a PRAM simulator peforms abysmally on some programs that step outside of the PRAM model by either explicitly or implicitly accessing a clock. This is analogous to the adversaries for universal hashing demonstrating that universal hashing performs abysmally on some programs that step outside of the reference-stream program model.

Also, note that having n processors simultaneously access the same memory location does not create a hot spot, as the requests will be combined in the network. To produce an $\Omega(n)$

slowdown, the adversary must determine $\Omega(n)$ distinct PRAM addresses that hash to the same memory of the target machine.

We first give a preliminary bound on the slow-down achievable by using an clocked adversary.

Theorem 4.1 *Let M be an n processor target machine. Then if $T(n) \geq n^2$, there is a clocked adversary that runs in time $O(T(n))$ on an n-processor PRAM and runs in expected time $\Omega(nT(n))$ under the hash-based PRAM simulation.*

Proof: Consider the following adversary. First, processor zero determines $n-1$ distinct addresses that are local to memory zero. Next, processor zero distributes these addresses, one to each of the remaining processors 1 through $n-1$. Finally, for $T(n)$ steps of the PRAM adversary program, processors 1 through $n-1$ ask for the address given them by processor 0. Below we show that this adversary achieves the bounds required by the theorem.

Because there are n memories, a random memory request by processor 0 has a $1/n$ chance of being local. Requesting a n^2 random memory locations will find n local addresses on average; this can be repeated k times to produce n distinct local addresses with high probability (where k depends only on the desired probability.) So in $O(n^2)$, processor 0 can discover $n-1$ local addresses.

If processors 1 throught $n-1$ all ask for their assigned addresses simultaneously, in each step of the simulation $n-1$ requests must enter and leave memory 0. (Because the addresses are distinct, no combining is possible.) Because the memory can only satisfy one request per time step, these requests cannot possibly be satisfied in fewer than $n-1$ steps of the target machine. Thus $T(n)$ such requests will take time $\Omega(nT(n))$, as required.

Finally, note that if $T(n) > n^2$, then the running time on the PRAM is $O(T(n))$, again as required in the statement of the theorem. □

The adversary in the proof of Theorem 4.1 is naive in that it makes no use of the specific family of hash functions used in the simulation. A more sophisticated attack, like those used in the adversaries for universal hashing, would allow the construction of adversaries that give linear slowdown for programs that have running times much lower than $O(n^2)$.

This adversary is also naive in that it makes no use of the parallel processing capability of the multiprocessor in discovering collisions. The parallel processing capability is used in the following lemma.

Theorem 4.2 *Let M be an n processor target machine. Then if $T(n) \geq n \log n$, there is a clocked adversary that runs in time $O(T(n))$ on a PRAM and runs in expected time $\Omega(nT(n))$ under the hash-based PRAM simulation.*

Proof: We prove the lemma by exhibiting an adversary that meets the bounds. The adversary makes note of the following property of the target machine: if processor i is accessing a local memory location at the same time as a remote processor is also accessing memory i, then processor i will see a slower response time than if no remote processor was accessing memory i, due to contention for the memory.

Consider the following adversary. First, processor 0 identifies a local memory location through $O(n)$ samples. Next, the remaining $n-1$-processors each generate a random address, and reference that address at the same time that processor 0 references its local address. If one of the $n-1$ random addresses is local to processor 0, then processor 0 will notice a slowdown, due to contention for the memory. If there is such a collision, the colliding address can be found in $O(\log n)$ PRAM steps by doing a binary search, that is, first processors 1 through $n/2$ access their addresses, then processors $n/2+1$ through n access their addresses. Then recur on the subset that contained a collision with processor 0's local address.

This can be repeated $c(n-1)$ times to identify $n-1$ addresses local to processor 0 in $O(n \log n)$ steps. After that, just as in the adversary in Theorem 4.1, processors 1 through $n-1$ access their assigned memory locations $T(n)$ times. □

5 Conclusion

Clocked adversaries constitute a class of programs that more closely model the capabilities of real programs than do such classes as reference-stream programs or PRAM programs. We have demonstrated that the performance of some hashing schemes must be re-examined when clocked adversaries are taken into account.

Clocked adversaries give rise to a number of open problems, including

1. Our adversaries for PRAM simulations ignore the important question of how large the queues in the interconnection network can grow during the simulation. How much can clocked adversaries degrade the queue lengths in PRAM simulations?

2. Are there efficient adversaries for hash functions of the form $h(x) = ((\sum_{0 \leq i \leq c} a_i x^i) \bmod p) \bmod n$?

3. Are there hashing functions that are immune to attacks from clocked adversaries? That is, are there hashing functions such that one can prove a lower bound on the amount of work a clocked adversary must do in order to degrade performance by a specified amount?

4. What is the expected performance of other types of data structures, including dynamic data structures (e.g. dynamic perfect hashing [DKM*88]) when faced with clocked adversaries?

One natural question is whether clocked adversaries represent a type of program people will actually write. After all, the general intent is that a program should try to use a data structure well, rather than as poorly as possible. In some sense, that is a moot point; theorems about the behavior of algorithms and data structures are intended to give users bounds on performance, and clocked adversaries illustrate that in some cases, the proof of the performance only holds if the program in question has no clock.

Also, there are many programs that do monitor their behavior and base decisions on measured performance. Such programs are especially prevalent in distributed or parallel enviroments. While the intent of such monitoring is to improve (rather than degrade) performance, clocked adversaries demonstrate how bad pathological cases can be.

References

[AS88] William C. Athas and Charles L. Seitz. Multicomputers: message-passing concurrent computers. *IEEE Computer*, 9–24, August 1988.

[CW79] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[DKM*88] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: upper and lower bounds. In *Proceedings of the 29th IEEE FOCS Conference*, pages 524–531, 1988.

[KU86] Anna Karlin and Eli Upfal. Parallel hashing — an efficient implementation of shared memory. In *Proceedings of the 27th Annual Symposium on Computer Science*, 1986.

[PN85] G. F. Pfister and V. A. Norton. Hot-spot contention and combining in multistage interconnection networks. In *Proceedings of the International Conference on Parallel Processing*, pages 790–797, 1985.

[Ran87] Abhiram G. Ranade. How to emulate shared memory. In *Proceedings of the 28th Annual Symposium on Computer Science*, pages 185–194, 1987.

[UW84] Eli Upfal and Avi Wigderson. How to share memory in a distributed system. In *Proceedings of the Sixteenth ACM STOC*, pages 171–180, 1984.