Space-Efficient Hot Spot Estimation[*]

Kenneth Salem

Institute for Advanced Computer Studies
and
Department of Computer Science
University of Maryland
College Park, Maryland, USA 20742

Abstract

This paper is concerned with the problem of identifying names which occur frequently in an ordered list of names. Such names are called hot spots. Hot spots can be identified easily by counting the occurrences of each name and then selecting those with large counts. However, this simple solution requires space proportional to the number of names that occur in the list. In this paper, we present and evaluate two *hot spot estimation* techniques. These techniques guess the frequently occurring names, while using less space than the simple solution. We have implemented and tested both techniques using several types of input traces. Our experiments show that very accurate guesses can be made using much less space than the simple solution would require.

[*]This work was supported by CESDIS, NASA's Center of Excellence in Space Data and Information Sciences

Space-Efficient Hot Spot Estimation

Abstract

This paper is concerned with the problem of identifying names which occur frequently in an ordered list of names. Such names are called hot spots. Hot spots can be identified easily by counting the occurrences of each name and then selecting those with large counts. However, this simple solution requires space proportional to the number of names that occur in the list. In this paper, we present and evaluate two *hot spot estimation* techniques. These techniques guess the frequently occurring names, while using less space than the simple solution. We have implemented and tested both techniques using several types of input traces. Our experiments show that very accurate guesses can be made using much less space than the simple solution would require.

1 Introduction

This paper is concerned with the problem of identifying names which occur frequently in an ordered list of names. Frequently occuring names are called *hot spots*. We are primarily concerned with on-line hot spot detection, in which the list may be scanned only once, in order.

Hot spot detection is not difficult. The obvious technique is to maintain a list of all of the names that occur in the input list, along with a counter for each name. Each time a name appears on the list, its counter is incremented. When the list is exhausted, those names with sufficiently large counter values are reported as hot spots.

The drawback of this technique is the amount of space it requires, particularly when the name space is large and the reference list is long. The space required is proportional to the number of unique names that appear in the input list. Yet, if we names to be hot spots if they appear with a frequency of at least T, then no more than $1/T$ names can possibly be hot spots, regardless of how many appear in the input.

Space is particularly important when reference counts are maintained in main memory, so that they can be searched and updated quickly. Fast, efficient hot spot detection algorithms are important in on-line applications in which the reference list is scanned "on-the-fly". For example, we may wish to monitor packets in a large network to determine which sources or destinations occur frequently. In this case, packet arrivals determine the reference list, and the hot spot detector must keep pace with them.

In this paper, we present and compare two hot spot estimation techniques. These techniques use less space than the simple technique described above. However, they do not guarantee a completely accurate solution. In some cases, they may inadvertantly report as hot a name that is not (a false positive). In others, they may fail to report a hot name (a false negative).

The two estimation techniques differ in the types of inaccuracies that they allow. The first technique guarantees that no false positives will be reported, i.e., any name that it reports as hot will truly be hot.

The second technique produces no false negatives. It correctly reports all hot spots, although it may inadvertently report some cold items as hot as well. We believe that each of these techniques has useful applications, and so we report them both here.

Hot spot estimation has many applications. We have used it in an adaptive storage manager to determine which stored data are accessed frequently [1, 2]. In this application, the ordered list of names is the stream of data requests arriving at the storage manager. The storage manager can take advantage of detected hot spots in a variety of ways. In our case, frequently accessed data is migrated to the center of a disk's platters to reduce expected seek times. Other performance-enhancing techniques, including prefetching, data replication and data declustering (in systems with multiple disks) can also be implemented adaptively using hot spot information.

Hot spot estimation can also be used as a data processing tool. For example, data classification can be accomplished using a simple clustering algorithm based on hot spots. The algorithm works as follows. First, the data are scanned to produce a (possibly multidimensional) histogram of values. Second, clusters are associated with peaks in the histogram. Finally, individual data points are assigned to clusters based on their proximity to the peaks (clusters). (An application of this technique for clustering of pixels in multi-spectral images is described in [12].) Clearly, the process of constructing a histogram to determine its peak values is hot spot detection. For data processing tasks such as this, it is not critical that on-line hot spot detection be used. However, since on-line techniques are fast and require relatively little memory, they may still be valuable when large volumes of data must be processed.

Hot spots are a well-known phenomenom in both computer systems and natural systems. Numerous studies, e.g., [4, 13, 5, 7]. have reported the existence of skewed data or request distributions in file systems and database systems. Generally, these studies have used off-line analyses of reference traces to measure the complete distributions. Analytic models, such as those based on the Zipf distribution, have also been developed to capture this skew.

Numerous, disparate techniques have been developed to take advantage of hot spot information. (Often it is assumed that the hot spots are known in advance.) For example, it is well known that disk seek times can be reduced by clustering hot data together on the disk. In data processing systems, special concurrency controls [6, 11] and data structures [5] can be used.

We are aware of one other paper [10] that addresses the hot spot detection problem. That paper presents two detection algorithms. Both detect hot spots with perfect accuracy, and one is known to have a space overhead proportional to $1/T$, where T is the frequency threshold for hot spots. However, both algorithms require two passes over the reference string, thus they cannot be used for on-line estimation.

Although they were not designed for this purpose, the first pass (only) of either algorithm from [10] could be used as an on-line estimation technique. Used this way, either algorithm could produce false positives, but no false negatives (like the RP algorithm described here). However, neither algorithm allows a tradeoff between space overhead and accuracy. In particular, if the estimate is inaccurate, there is no mechanism to improve it by using additional space. We will not pursue either of these techniques further here.

The next section of this paper describes the two hot spot estimation techniques that we have considered. In Section 3 we define and prove several useful properties of these algorithms. We have implemented and evaluated both techniques using reference traces drawn from several application domains. In Section 4 we describe the traces and the results of our evaluation. Our tests demonstrate that, in practice, both techniques can provide very accurate hot spot estimates using a fraction of the space that would be required to learn the entire request distribution with perfect accuracy.

2 Hot Spot Estimation

A hot spot estimation algorithm scans a sequence of names, called the *reference string.* When the scan is completed, the algorithm reports those names that appeared frequently during the scan. These names are the hot spots detected by the algorithm.

Hot spots are defined as names which appear with a frequency of at least T_H in the reference string. The value T_H is called the *hot temperature threshold.* It is a parameter to the hot spot detection algorithm. For our algorithms, threshold values are specified as positive real numbers no greater than one. A name is a hot spot if it occurs at least LT_H times in the reference string, where L is the length of the string.

In the remainder of this section, we will describe two different hot spot estimation algorithms. For each algorithm, we present an both an informal description and an implementation in pseudo-code. Each implementation is defined by three programs: *Initialize*, *ProcessReference*, and *ReportResults.* The *ProcessReference* routine is called once for each name that is scanned from the reference string. The *Initialize* and *ReportResults* routines are called at the beginning and end of the scan, respectively.

2.1 The Name Cache Algorithm

The *Name Cache* (NC) algorithm maintains a list of names from the reference string and counts references to them. As each name is scanned from the reference string, its reference count is incremented if it is already present in the name cache. If the referenced name does not appear in the cache and the cache is not full, the new name is cached with an initial reference count of one. If the cache is full, NC uses a replacement heuristic to guide its behavior. The replacement heuristic selects one name to be eliminated from the cache. The newly-referenced name is then added to the cache with a reference count of one.

We have considered three replacement heuristics. Each heuristic selects one eligible cached name to be removed (along with its reference count) from the cache. The three replacement heuristics are:

LRU The least recently used eligible name on is removed from the cache.

Random An eligible name is selected randomly for removal. All eligible names are equally likely to be selected.

Biased An eligible name is selected randomly for removal. A name's selection probability is chosen to be inversely related to its reference count. Specifically, if eligible names i and j have reference counts c_i and c_j, then their probabilities of selection (p_i and p_j), satisfy

$$\frac{p_i}{p_j} = \frac{c_j}{c_i}$$

A name's eligibility for replacement is determined by a screening process that uses a parameter called the *eligibility threshold*, T_{elig}. A cached name is considered eligible for replacement if its current estimated temperature is below T_{elig}. A name's estimated temperature is simply its current reference count divided by the total number of references that have been scanned. The eligibility threshold T_{elig} is a tunable parameter to the NC algorithm.

When a replacment is required, it is possible that no cached names will be eligible. In that case, no replacement occurs. The just-referenced name is simply ignored, and is not added to the name cache.

Figure 1 shows the *Initialize*, *ProcessReference*, and *ReportResult* routines that implement the NC algorithm. The algorithm takes two parameters in addition to the hot temperature threshold T_H. These parameters are the safety threshold T_{elig} and the maximum size of the name cache.

2.2 The Random Partitioning Algorithm

The second hot spot detection technique is called *random partitioning* (RP). Under RP, hash functions are used to partition the name space. A single reference counter is used to count references to all names that fall into a single partition. The RP algorithm simultaneously maintains several distinct partitionings of the name space. Names are reported as hot spots only if they fall into frequently referenced paritítions under *all* of the partitionings. A pseudo-code implementation of RP is shown in Figure 2.

Suppose that hash functions are available which map names to integers in the range $[1, C]$. Each such function defines a partitioning of the name space into C distinct subspaces. The RP algorithm uses K such functions to define K different partitionings of the name space. Every name is assigned to exactly K partitions, one for each of the K hash functions.

The RP algorithm maintains C reference counters for each of K hash functions. We will use f_i to denote the ith hash function, and $counter_i[j]$ to denote the jth reference counter for hash function f_i A reference to name n causes $counter_i[f_i(n)]$ to be incremented (for each $1 \leq i \leq K$). A name n is reported as a hot spot if all of the partitions (counters) it hashes to are hot.

2.2.1 The Candidate Set

One difficulty with the RP algorithm as described is that it does not store any of the names from the input sequence. Unlike the NC algrithm, RP does not associate each reference counter with a single name. Once the reference string has been scanned, RP's reference counters can be used to test whether a given name n is (estimated to be) hot. However, RP has no way of knowing which names to test. An exhaustive check of the name space is likely to be very time consuming, or impossible if the name space in infinite.

To avoid this problem, RP maintains an additional data structure called the candidate set. The candidate set is a set of names whose temperatures should be tested after the reference string has been completely scanned. Names that are tested and found to be hot are reported as hot spots by the algorithm.

The RP algorithm uses two rules to manage the candidate set.

```
INPUT PARAMETERS USED:
$T_H$: the hot temperature threshold
$T_{elig}$: the replacement eligibility threshold
$MaxCacheSize$: maximum number of cache entries allowed

DATA STRUCTURES:
$Cache$: a set of entries, where each entry $X$ includes a name ($X.name$) and a reference count ($X.count$)
$RefCount$: an integer, the total number of names scanned from the input so far

PROCEDURE Initialize()
BEGIN
    $Cache \leftarrow \emptyset$
    $RefCount \leftarrow 0$
END

PROCEDUREProcessReference(name)
BEGIN
    $RefCount \leftarrow RefCount + 1$
    look for an entry $X$ in $Cache$ such that $X.name = name$
    IF there is such an entry $X$ THEN
        $X.count \leftarrow X.count + 1$
    ELSE IF $|Cache| < MaxCacheSize$ THEN
        add a new entry to $Cache$, with name $name$ and reference count one
    ELSE
        let $E = \{X | X \in Cache \wedge \frac{X.count}{RefCount} < T_{elig}\}$
        IF $E \neq \emptyset$ THEN
            select $Y \in E$ according to replacement policy
            delete $Y$ from $Cache$
            add a new entry to $Cache$, with name $name$ and reference count one
END

PROCEDURE ReportResult()
BEGIN
FOREACH entry $X$ in $Cache$ DO
    IF $\frac{X.count}{RefCount} \geq T_H$ THEN OUTPUT($X.name$)
ENDFOREACH
END
```

Figure 1: Pseudo-Code for the NC Algorithm

```
INPUT PARAMETERS USED:
$T_H$: the hot temperature threshold
$K$: the number of hash functions used
$C$: size of the range of each hash function

DATA STRUCTURES:
$Counters[K][C]$: a $K$-by-$C$ array of reference counters
$CandidateList$: a set of names
$RefCount$: the total number of names scanned from the input so far

PROCEDURE Initialize()
BEGIN
    FOREACH $1 \leq i \leq K$ and $1 \leq j \leq C$ DO $Counters[i][j] \leftarrow 0$
    randomly select $K$ hash functions $f_i$, $1 \leq i \leq K$
    $RefCount \leftarrow 0$, $CandidateList \leftarrow \emptyset$
END

PROCEDURE ProcessReference(name)
BEGIN
    $RefCount \leftarrow RefCount + 1$, $min \leftarrow \infty$
    FOREACH $i$ FROM 1 TO $K$ DO
        $Counters[i][f_i(name)] \leftarrow Counters[i][f_i(name)] + 1$
        IF $(min > Counters[i][f_i(name)])$ THEN $min \leftarrow Counters[i][f_i(name)]$
    END FOREACH
    /* $min$ now contains minimum reference count of all partitions to which $name$ hashed */
    IF $(\frac{min}{RefCount} \geq T_H)$ THEN
        $CandidateList \leftarrow CandidateList \cup \{name\}$
END

PROCEDURE ReportResult()
BEGIN
FOREACH name $N$ in $CandidateList$ DO
    $min \leftarrow \infty$
    FOREACH $1 \leq i \leq K$ DO IF $(min > Counters[i][f_i(N)])$ THEN $min \leftarrow Counters[i][f_i(N)]$
    IF $(\frac{min}{RefCount} \geq T_H)$ THEN OUTPUT($N$)
END
```

Figure 2: Pseudo-Code for the RP Algorithm

Insertion Rule: When a name n is referenced, it is inserted into the candidate set if its current estimated temperature is hot, i.e., above T_H. (No action is required if n is already in the set.)

Deletion Rule: A name may be removed from the set only if its current estimated temperature is not hot.

The estimated temperature of a name is simply the minimum of the reference counts of the K counters the name hashes into, divided by the total number of references so far. As we will discuss in the next section, these rules are sufficient to ensure that all truly hot names will be in the candidate set after the reference string has been completely scanned.

The Deletion Rule permits some latitude in the timing of deletions. An very aggressive implementation might check the current temperatures of candidate names very frequently so that cooling names do not remain in the candidate list any longer than necessary. A lazier implementation could check less frequently for deletion. In fact, it is never necessary to delete names from the set at all.

Our implementation of RP uses an opportunisitc approach that lies in between these extremes. The candidate set is implemented using a hash table. During a lookup or insert into the table, any names that are touched have their current temperature checked. Names are deleted if they are found to be not hot.

2.2.2 Hashing

The accuracy of the RP algorithm depends on the choice of hash functions (f_i) used to partition the name space. Consider two names x and y, one hot and the other cold. If $f_i(x) = f_i(y)$ for all i, then the two names lie in all of the same parititions, and RP will be unable to distinguish between them. In other words, both x and y will be reported hot.

To reduce the likelihood of this occurrence, RP's hash functions are randomly generated from a class of *universal*$_2$ hash functions [3]. A *universal*$_2$ hash function class has the following useful property: no pair of distinct names collides under more than $1/C$th of the functions in that class. (As used above, C is the size of the range of the hash functions.) In other words, by increasing the number of hash functions used, we make it increasingly likely that any pair of names can be distinguished.

Our implementation of RP uses hash functions from the *universal*$_2$ class of the form

$$h_{a,b}(x) = ((ax + b) \bmod p) \bmod C$$

where p is a large prime number (preferably larger than the size of the name space) and $a, b \in \{0, 1, \ldots, p-1\}, a \neq 0$. Functions from this class are chosen by random selection of the constants a and b.

These hash functions can be used directly if the referenced names are integers. If the names are character strings (or some other type), as was the case for one of our test traces, they must be converted to integers before hashing. Thus, if the string to integer conversion function is denoted f_{conv}, the actual hashing functions used are of the form $h_{a,b}(f_{conv}(x))$.

Unfortunately, any names that collide under f_{conv} can never be resolved by the RP algorithm. Several common, simple conversion functions produced significant numbers of collisions in our test traces. (For example, combining character codes using exclusive-or to produce an integer key, as suggested in [8], did not work well on the file names in our NCAR trace.) We were able to reduce this problem by using a different conversion function for each integer hash function $h_{a,b}$. Specifically, we used a general conversion function which accepted the constants a, b as parameters, giving us string hashing functions of the form $h_{a,b}(f_{conv}(x, a, b))$. Although collisions are still possible with such a function, it is less likely that a pair of names will collide under K randomly selected functions.

3 Properties of the Algorithms

The two hot spot estimation algorithms have complementary properties. The NC algorithm never produces false positives, i.e., it never mistakenly reports a cold name as hot. Under certain conditions, which we will describe shortly, the RP algorithm does not produce false negatives. In this section we show why these properties hold.

Throughout the section, we will use the following notation. We will denote by $R(n, i)$ the number of references to name n among the first i references of the reference string. The *temperature* of name n after i references, $T(n, i)$ is given by

$$T(n, i) = \frac{R(n, i)}{i}$$

We will use $T(n)$ as a shorthand for $T(n, L)$, where L is the length of the reference string. This is called the *final temperature* of name n.

Each of the hotspot algorithms uses reference counters to estimate the true reference count of each name. Since the algorithms work with limited numbers of counters, the estimates they produce may not be accurate. We will use $\tilde{R}(n, i)$ to denote an algorithm's estimated reference count for n after i references have been processed. Estimated temperature, $\tilde{T}(n, i)$, is defined by the ratio of $\tilde{R}(n, i)$ to i, as above.

3.1 The Name Cache Algorithm

The estimated reference counts for the NC algorithm can be defined as follows. If name n is in the name cache after i references have been processed, then $\tilde{R}(n, i)$ is equal to the value of n's reference counter. If n is not in the cache after i references, then $\tilde{R}(n, i)$ is defined to be zero. We can now prove the following simple lemma, which says that the name cache algorithm never overestimates reference counts.

Lemma 3.1 *For all names n and for all $0 < i \leq L$, $R(n, i) \geq \tilde{R}(n, i)$ under the NC algorithm.*

Proof: This can be shown by induction on i. Clearly, $R(n, 1) = \tilde{R}(n, 1)$. Assume that $R(n, i-1) \geq \tilde{R}(n, i-1)$. If the ith reference is to n, then $R(n, i) = R(n, i-1) + 1$. If n was in the name cache, the estimated count for n will also increase by one, otherwise it will be exactly one. In either case, the Lemma holds. If the ith reference is not to n, then $R(n, i) = R(n, i-1)$. The estimated count either remains the same, or becomes zero if n is replaced in the cache. In either case, the Lemma holds. $\Box$

The NC algorithm reports a name n as hot if its final estimated temperature $\tilde{T}(n)$ is greater than the hot temperature threshold T_H. Since $T(n) \geq \tilde{T}(n)$ (from Lemma 3.1), any name reported hot by NC must truly be hot. However, NC may fail to report a hot name by severely underestimating that name's reference count. Clearly, the greater the size of the name cache, the less likely the algorithm will be to report false hot spots. We explore this trade-off between space and accuracy in Section 4.

3.2 The Random Partitioning Algorithm

The RP algorithm uses K hash functions to paritition the name space. The estimated reference count of a name n is the minimum of the counts of the K counters that n hashes to. Thus, if $counter_i(j)$ is the value of the jth counter for the ith hash function, then

$$\tilde{R}(n,i) = \min_{1 \le k \le K} counter_k(f_k(n))$$

The following lemma states that the RP algorithm never underestimates reference counts.

Lemma 3.2 *For all names n and for all $0 < i \le L$, $R(n,i) \le \tilde{R}(n,i)$ under the RP algorithm.*

Proof: By induction on i. For $i = 1$, either the first reference to n or it is not. If it is to n, then $R(n,1) = \tilde{R}(n,1) = 1$. Otherwise, $R(n,1) = 0$ and $\tilde{R}(n,1)$ is either zero or one. Assume $R(n,i-1) \le \tilde{R}(n,i-1)$. If the ith reference is to n, then $\tilde{R}(n,i) = \tilde{R}(n,i-1)+1$ since each $counter_k(f_k(n))$ will be incremented by the algorithm. The true reference count for n also increases by one. If the ith reference is to $m \ne n$, then $R(n,i) = R(n,i-1)$ and the estimated reference count either remains unchanged or increases by one. (An increase by one occurs if there is a hash collision between n and m under any of the hashing functions, i.e., if $f_k(n) = f_k(m)$ for any $1 \le k \le K$.) In either case, the Lemma holds. □

To make claims about the hot spots reported by RP, we must also model the state of the candidate set. We will let $C(i)$ represent the state of the candidate set after any insertions or deletions resulting from the first i references have been performed. The next lemma says that the candidate set must include all hot names from the reference string.

Lemma 3.3 *For all names n and for all $0 < i \le L$, $(T(n,i) \ge T_H) \Rightarrow (n \in C(i))$, under RP.*

Proof: By induction on i. After one reference, all names have a true temperature of zero except the referenced name, which has a temperature of 1.0. That is the only name which can satisfy the antecedent of the Lemma, and it must be inserted into the candidate set because of the Insertion Rule. Assume that $(T(n,i-1) \ge T_H) \Rightarrow n \in C(i-1)$. Suppose that n satisfies the antecedent of the Lemma after reference i. If $n \in C(i-1)$ then $n \in C(i)$, since $\tilde{T}(n,i) \ge T(n,i) \ge T_H$ and the Deletion Rule prevents it from leaving the candidate set. If $n \notin C(i-1)$, then $T(n,i-1) < T_H$ by the inductive hypothesis. Since $T(n,i) \ge T_H$, the ith reference must have been to n, since a name's true reference count (and true temperature) can only increase when it is referenced. Since $\tilde{T}(n,i) \ge T(n,i)$, the Insertion Rule forces n into $C(i)$. □

By Lemma 3.3, all hot names will appear in the candidate set after the references string has been scanned. By Lemma 3.2, the final estimated temperatures of all such names will be hot, and therefore they will be reported by the algorithm. Thus, the RP algorithm will never fail to report a hot name.

Lemma 3.1 holds regardless of the number of reference counters used. Thus, regardless of the amount of space used for hot spot detection under the NC algorithm, no false hot spots will be reported. Unfortunatly, it is difficult to make an analogous claim for the RP algorithm. Although Lemma 3.2 holds regardless of the number of reference counters used, Lemma 3.3 depends on a candidate set managed using the Insertion Rule and the Deletion Rule. In general, it is not possible to set an upper limit[†] on the size of the candidate set while still observing the rules. The problem occurs if a new hot name must be added to the set when it is filled to capacity with other hot names.

Thus, if the size of the candidate set is fixed in advance, it is possible for the RP algorithm to fail to report a truly hot name. However, the following claim is possible: if there are no violations of the Insertion Rule or the Deletion Rule during processing of the reference string, then the RP algorithm will not fail to report any true hot spots. In other words, although RP cannot always avoid false negatives when the candidate set size is limited, it has a mechanism for determining whether false negatives are possible after processing a given reference trace.

4 Performance Analysis

We have implemented and and evaluated both hot spot detection algorithms. Our evaluation was performed using a set of input sequences from the application problem domains mentioned in the introduction. The goals of our evaluation were, first, to determine the effectiveness of each algorithm when applied to realistic input sequences, and second, to determine how best to use the algorithms. In particular, we were interested in how to set the algorithm-specific parameters.

4.1 Input Data

Both algorithms were evaluated using three input sequences representing different application domains. The input sequences included the following:

UMD A trace of physical disk block references obtained from a Unix file server in the University of Maryland's Computer Science Department.

NCAR A trace of file transfers to and from a mass storage system at the National Center for Atmospheric Research.

NASA A raster scan of pixel intensities from a four-band multispectral Landsat image.

A summary of some of the characteristics of these traces is given in Figure 3.

The UMD trace was collected from a Fujitsu 2351 Eagle disk attached to a Sun workstation running version 3.2 of SunOS. Each trace entry corresponds to a read or write request sent to the disk's driver. The sequence of requested disk addresses (block numbers) forms the input sequence used to drive the hot

[†]Of course, the number of names that can be truly hot at any time is bounded from above by $1/T_H$. However, the Insertion and Deletion rules are specified in terms of estimated temperatures, not true temperatures. The number of estimated hot spots may be much greater than $1/T_H$, particularly if very few reference counters are available.

Trace Name	Trace Length	Number of Unique Names	Temp. of Hottest Name	Temp. of 10th Hottest Name	Temp. of 100th Hottest Name	Temp. of 1000th Hottest Name
UMD	93664	6045	0.0529	0.0079	0.0015	0.0001
NCAR	94633	30667	0.0036	0.0020	0.0008	0.0002
NASA	262144	11986	0.0064	0.0030	0.0015	0.0001

Figure 3: Characteristics of Input Traces

spot detection algorithms. A total of 45138 unique block numbers are possible with this disk. However, Figure 3 shows that the actual number of unique block numbers occurring in the trace was much smaller.

The traced disk held a file system containing primarily shared binary files. The trace reflects requests for these files generated by multiple concurrent processes running on networked workstations. It covers a period of approximately sixteen hours, from 10am until 2am, on a weekday.

The NCAR trace captures one month of file transfers between the NCAR mass storage system and its supercomputing center. The trace was recorded during November, 1990. Each trace entry specifies a fully-qualified file name within the mass-storage system. The sequence of file names from the trace is the input sequence used to drive the hot spot detection algorithms. Further information on the collection of these data and on the NCAR mass storage system itself can be found in [9].

The NASA trace is intended to be representative of an input sequence from a scientific data processing application. The trace consists of a list of pixel values taken from a multi-spectral image of the Washington, DC metropolitan area. The image was produced from data acquired by the Landsat thematic mapper (TM) instrument.

The image includes data from four spectral bands. Thus, each pixel value consists of a four-tuple of intensity values, one from each of the spectral bands. Intensity values for each band are quantized to seven bits. The trace was derived from a row-major scan of the pixels in the full $512x512$ pixel image, resulting in an input sequence 256K entries long. The input sequence used to drive the hot spot detectors is the sequence of four-tuples from the trace. Application of a hot spot detection algorithm to the NASA trace produces a list of frequently occurring intensity tuples. Such a list has useful applications. For example, it can be used to define potential clusters (classes) in an unsupervised classification if the image's pixels.

4.2 Methodology

To determine the performance of the hot spot algorithms, we computed the true reference frequency (temperature) of each name in each of our test traces. These frequencies are used to determine the true hot set for each trace, as a function of the hot temperature threshold. The true frequencies were determined by applying the NC algorithm using an unlimited cache size, so that replacement of names from the cache was never necessary.

In each of the following experiments, we compare the size of the hot set predicted by an algorithm to the size of the true hot set. A hot set's size is simply the number of names that it contains. This simple

Algorithm	Symbol	Parameter Meaning	Default Value
Name Cache	T_{safe}	safety threshold	0.0003
Name Cache	-	replacement policy	biased
both	T_H	hot temperature threshold	0.002
Random Partitioning	K	number of hash functions	3

Figure 4: Default Algorithm Parameter Settings

metric does not provide information on which hot names are not reported (under NC) or which cold names are mistakenly reported (under RP). Our simulator did maintain additional, weighted metrics in an attempt to capture more detailed information. However, we found that these more complex metrics did not provide much additional insight in the performance of the algorithms. Thus, we have reported only the simple metric here.

4.3 Accuracy vs. Number of Counters

In our first set of experiments, we sought to determine the accuracy of the hot spot detection algorithms as a function of the number of reference counters used. All other algorithmic parameters were fixed at default values, which are summarized in Table 4. The effects of these parameters are considered in later later experiments. For the RP algorithm, the candidate set size was unbounded. We discuss the candidate set further in Section 4.5.

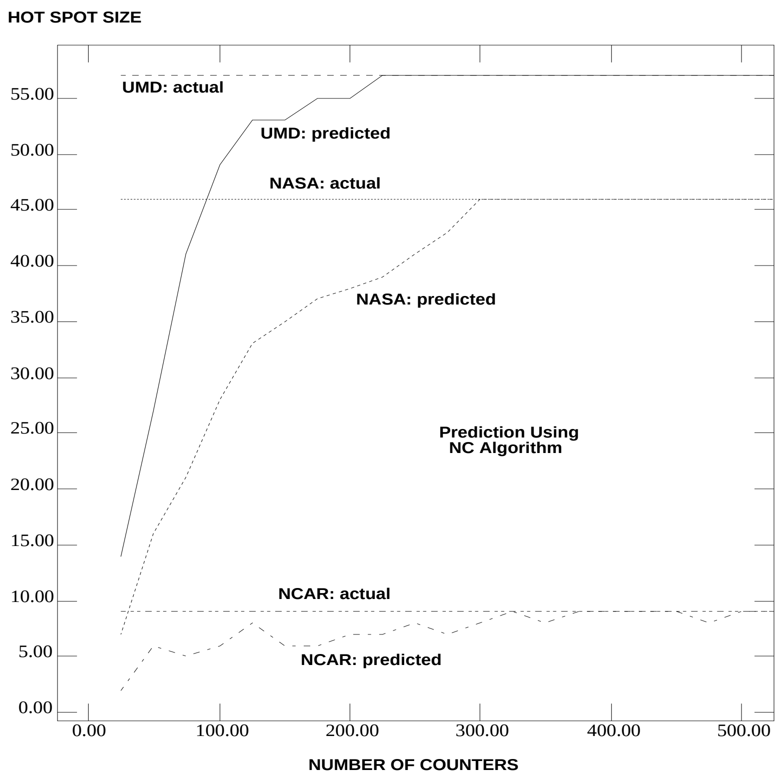
Figure 5 summarizes the performance of the two algorithms on all three of the input traces. For each trace, the size of the true hot spot (for $T_H = 0.002$) is shown, along with the size of hot spot detected by the algorithm. Since the NC algorithm underestimates temperatures, its hot spot prediction converges to the true hot spot from below as the number of reference counters in increased. Conversely, the RP algorithm overestimates the hot spot size and its prediction converges from above. In all cases, the detected hot spot converges to the true hot spot as the number of reference counters is increased.

Comparison of the two graphs shows that the RP algorithm requires more counters (by about an order of magnitude) than NC to function effectively. However, in both cases the number of counters required for a very accurate estimate of the hot spot is significantly less than the number of unique names appearing in the input traces (see Figure 3).

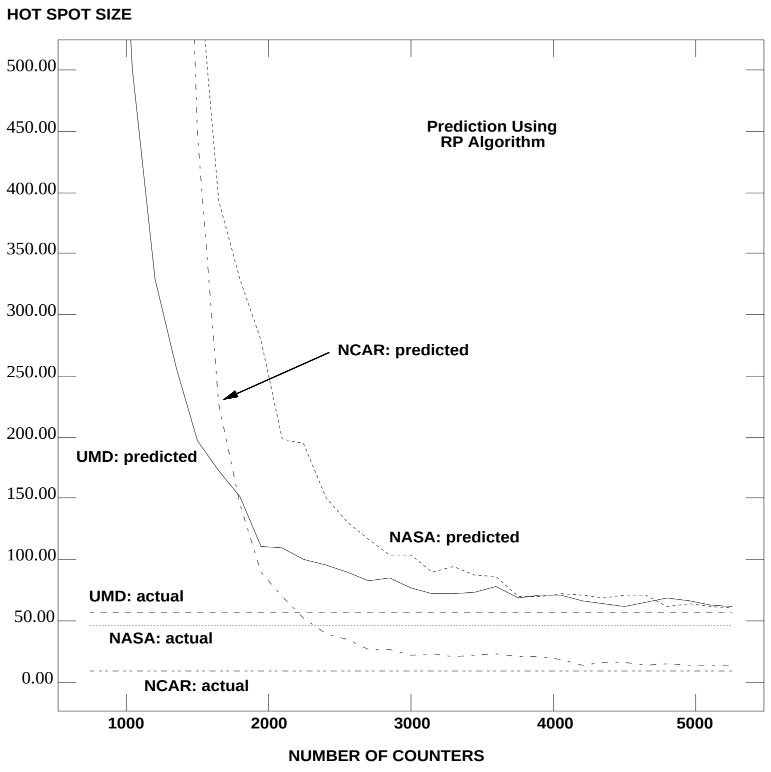
4.4 Effect of the Hot Temperature Threshold

As T_H is varied, the size of the true hot spot changes. We expect that larger values of T_H (smaller hot spot sizes) will require fewer reference counters for accurate prediction. In our next experiment we varied T_H to test this hypothesis. Figure 6 shows the results of this experiment for the UMD trace. (The remaining traces produced similar results.) Algorithm-specific parameters remained fixed at the default values shown in Figure 4.

As expected, more counters are required to accurately predict the hotspot when T_H is small. For example, the RP algorithm produces a very accurate hot spot estimate using 1000 counters when $T_H =$

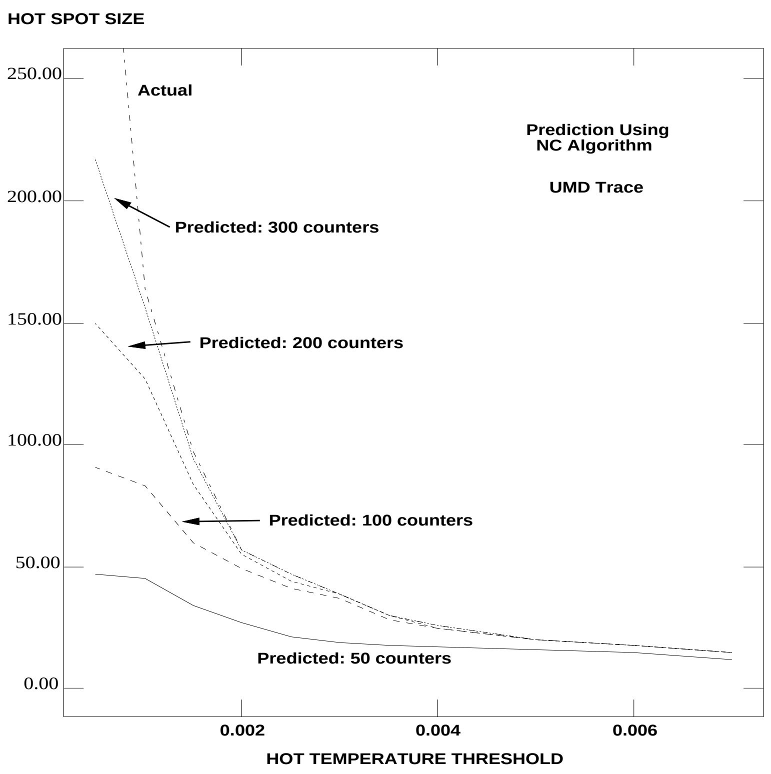


(NC algorithm)

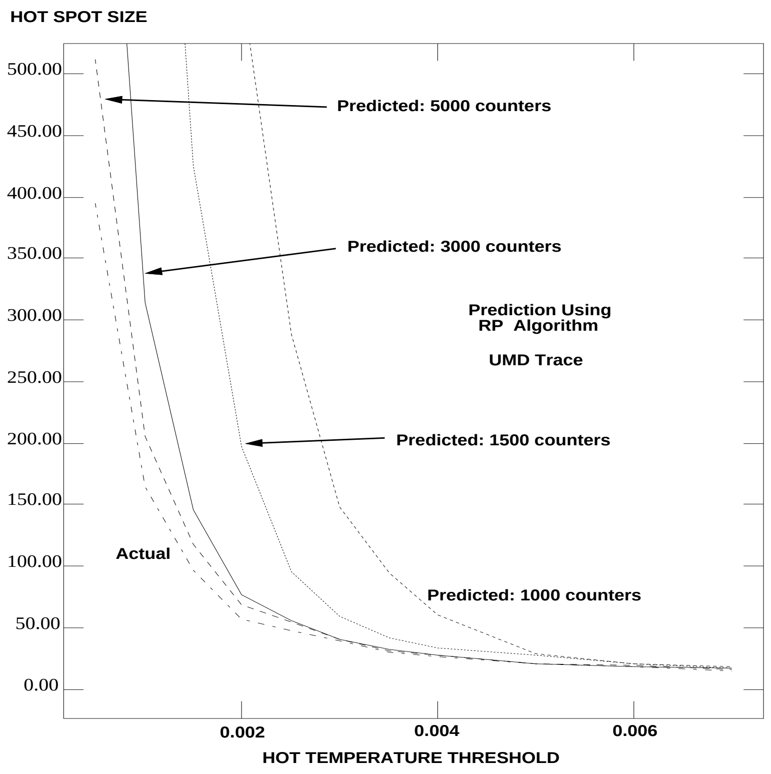


(RP algorithm)

Figure 5: Predicted and Actual Hot Spot Sizes vs. Number of Reference Counters



(a)



(b)

Figure 6: Predicted and Actual Hot Spot Sizes vs. Hot Temperature Threshold

0.006. However, more than 5000 counters are required to produce comparable accuracy when $T_H = 0.002$. The figure also illustrates the declining marginal benefit of additional reference counters as the number of counters increases. Given a particular value of T_H, the marginal benefit of using additional counters can be seen by drawing a vertical line through T_H on the figure's horizontal axis.

It would be desirable to be able to predict in advance how many reference counters would be required for an accurate hot spot diagnosis. Unfortunately, the relationship between the number of reference counters, the value of T_H, and the accuracy of the prediction is very complex. Furthermore, we have found that this relationship is trace-dependent.

Nonetheless, Figure 6 does suggest an iterative method for determining an appropriate number of counters to use. The basic idea is to run the detection algorithm several times, either on the same trace or on a series of similar traces. Before each successive run, the number of counters to be used is increased. This process is repeated until the size of reported hot spot does not change significantly from run to run. Of course, a more sophisticated search strategy, such as binary search, can be used to reduce the number of runs required. We have successfully applied this technique to a series of traces of disk accesses (similar to the UMD trace).

4.5 Candidate List Size

The RP algorithm maintains a list of candidate names, in addition to its reference counters. This represents a space overhead over and above the space used by reference counters.[‡] Figure 7 shows the number of candidates in the set as a function of the number of references processed, for varying numbers of reference counters. These data were obtained using the UMD trace and the default parameter values from Figure 4.

Two points can be observed from this figure. First, if too few reference counters are used, the candidate list can continue to grow as the reference list is processed. However, we have found that this occurs only if so few reference counters are used that the final predicted hot set will not be very accurate. Such behavior can be avoided if the number of reference counters is selected properly. In fact, a long-term increase in the size of the candidate list can serve as a indication that insufficient counters are being used for accurate prediction.

Second, the candidate list is largest while the very beginning of the reference stream is being processed (assuming enough reference counters are being used). Unfortunately, *all* names that appear very early in the reference string will have initial temperatures that are high relative to T_H. For example, names that appear in within the first $1/T_H$ references will have a temperature greater than T_H when first referenced, and thus will be placed in the candidate list. This effect accounts for the initial transient that can be observed in Figure 7. It is possible to avoid this transient by simply not adding any names to the candidate list early in the reference string. While this may be acceptable in practice, it may lead to a violation of Lemma 3.3, which guarantees that RP will not miss any truly hot names.

[‡]The NC algorithm must also store names. However, the number of names stored is the same as the number of reference counters used.

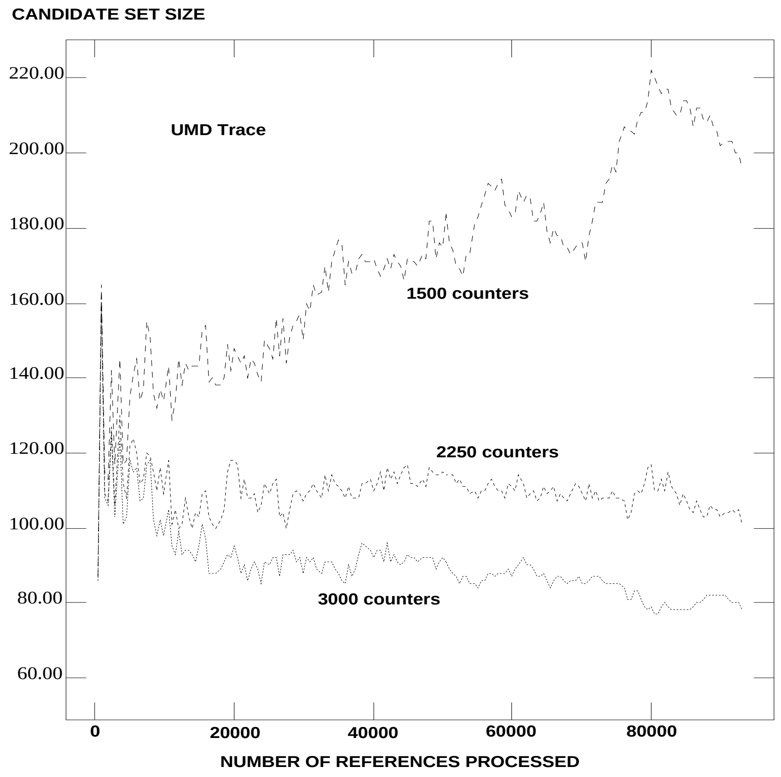


Figure 7: Size of the Candidate List vs. Number of References Processed

4.6 Effects of Algorithm-Specific Parameters

4.6.1 Effect of the Eligibility Threshold

The NC algorithm uses an eligibility threshold T_{elig} to select eligible names for replacement in the cache. Figure 8 illustrates the effect of T_{elig} on the performance of NC. Two graphs are shown, one for $T_H = 0.001$ and one for $T_H = 0.004$.

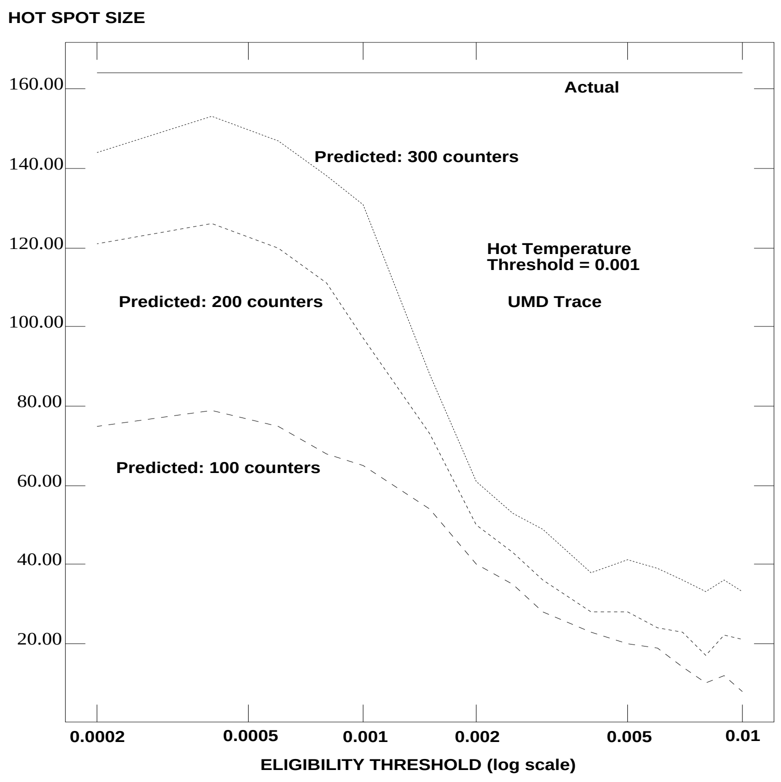
As the value of T_{elig} is increased, its impact diminishes because fewer names will be deemed ineligible for replacement. Figure 8 shows that the safety threshold is important: the algorithm performs poorly when T_{elig} is too high. As Figure 8 illustrates, the ideal value of T_{elig} depends somewhat on T_H. We have found that a good rule of thumb is to set T_{elig} about an order of magnitude less than the desired T_H. Starting from this value, the safety threshold can then be fine-tuned for a particular type of trace.

The optimal safety threshold does not depend strongly on the number of reference counters used. However, when there are many counters the range of safety thresholds that gives good performance widens.

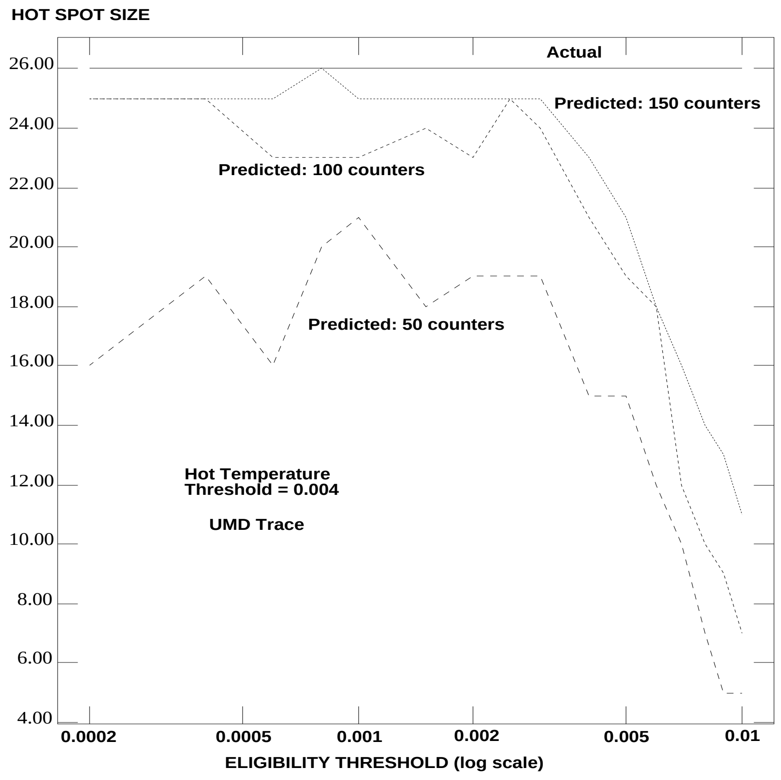
4.6.2 Effect of the Replacement Policy

Figure 9 compares the three replacement policies for the NC algorithm, using the UMD trace. This figure was produced using the default hot temperature threshold value of 0.002.

The replacement policy does not have a significant impact on the algorithm's performance. Across



(a)



(b)

Figure 8: Predicted and Actual Hot Spot Sizes vs. Safety Threshold

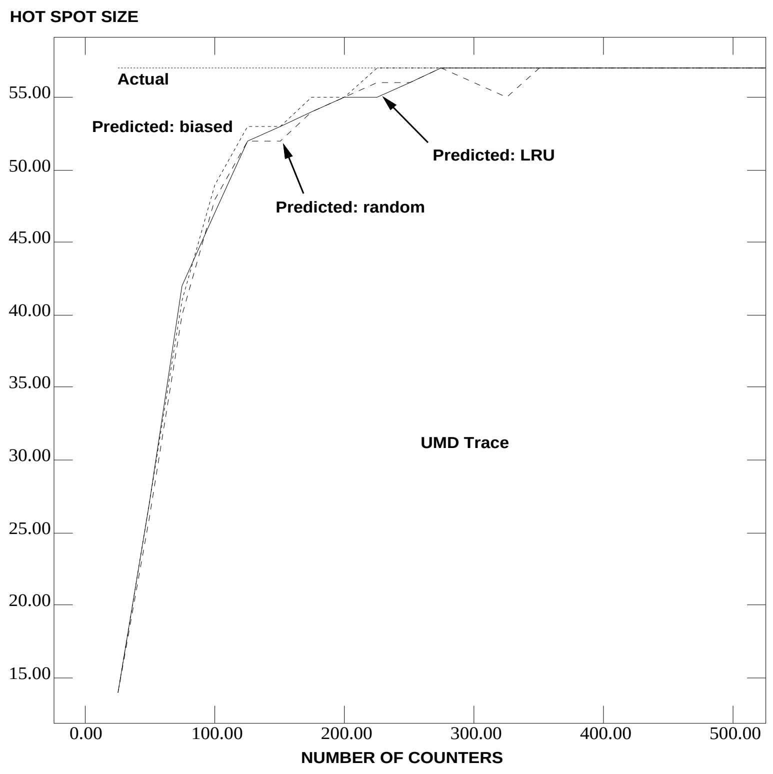


Figure 9: Predicted and Actual Hot Spot Sizes vs. Replacement Policy

all three traces, we found that the biased policy was somewhat better than the other two. However, the differences were not large. Overall, the pre-screening accomplished by the eligibility threshold (Figure 8) is much more important than the replacement policy in the NC algorithm.

4.6.3 Effect of the Number of Hash Functions

Figure 10 shows the effect of varying the number of hash functions used by the RP algorithm, while keeping the total number of reference counters constant. Thus, as the number of hash functions increases, the range of each function must shrink. Other parameters are set to their default values.

The use of a small number (greater than one) of hash functions produces the best performance. When sufficient reference counters to permit accurate predictions are used, two to four functions produced the best performance. Furthermore, performance drops off only slowly as the number of functions is increased. We found this conclusion to be insensitive to T_H. However, as we have shown previously, a greater total number of counters is required for accurate prediction when T_H decreases.

When too few counters are available for accurate prediction (e.g., 2000 counters in Figure 10), the number of hash functions used is more critical. Using exactly two hash functions provides the best performance possible under these conditions.

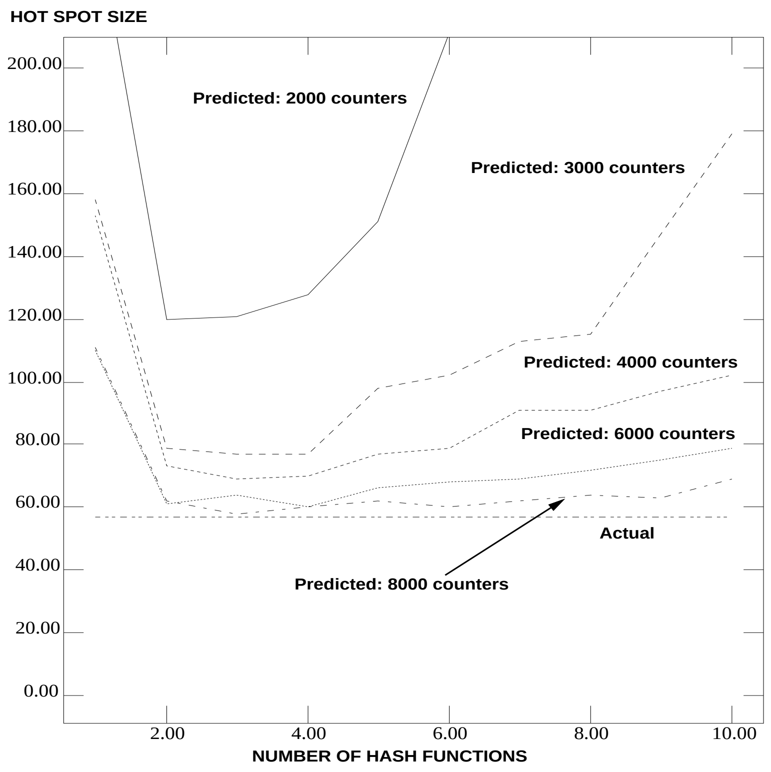


Figure 10: Predicted and Actual Hot Spot Sizes vs. Number of Hash Functions

5 Conclusion

We have described and evaluated NC and RP, two algorithms for on-line estimation of hot spots. These algorithms may make errors in their estimates. However, they require little space. Furthermore, it is possible to trade additional space for greater accuracy. The goal of our performance evaluation was to evaluate this tradeoff.

Either algorithm can detect hot spots accurately using much less space than a simple algorithm which counts references to all names in the input string. At our default parameter settings, the NC algorithm required only a few hundred reference counters to produce very accurate hot spot estimates from our traces, which contained tens of thousands of unique names. Typically, the RP algorithm required about an order of magnitude more space to produce good estimates. However, RP is guaranteed not to neglect any true hot spots in its estimate.

For either algorithm, the space required for accurate estimation depends strongly on the hot spot threshold temperature. Lower thresholds result in larger hot spots and require more space for accurate estimation. In addition, both algorithms include tunable parameters. Our evaluation provided guidelines for setting them.

Implementations of both algorithms, in C, are available from the author.

6 Acknowledgements

Bongki Moon implementated preliminary versions of these algorithms. The author is grateful to Ethan Miller, from the University of California at Berkeley, for providing the NCAR trace.

References

[1] Sedat Akyürek and Kenneth Salem. Adaptive block rearrangement. In *Proceedings of the International Conference on Data Engineering*, pages 182–189, April 1993.

[2] Sedat Akyürek and Kenneth Salem. Adaptive block rearrangement in unix. In *Proc. Usenix Summer Conference*, pages 307–321, June 1993.

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

[4] S. Christodoulakis. Implication of certain assumptions in data base performance evaluation. *ACM Transactions on Database Systems*, June 1984.

[5] Christos Faloutsos and H. V. Jagadish. On B-tree indices for skewed distributions. In *Proceedings of the International Conference on Very Large Data Bases*, pages 363–374, August 1992.

[6] Dieter Gawlick and David Kinkade. Varieties of concurrency control in IMS/VS Fast Path. *Database Engineering*, 8(2):3–10, June 1985.

[7] David W. Jensen and Daniel A. Reed. File archive activity in a supercomputing environment. In *Proceedings of the International Conference on Supercomputing*, July 1992.

[8] Donald E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[9] Ethan L. Miller. File migration on the Cray Y-MP at the National Center for Atmospheric Research. Technical report, Computer Science Division, Dept. of EECS, University of California at Berkeley, June 1991.

[10] J. Misra and D. Gries. Finding repeated elements. *Science of Computer Programming*, 2(2):143–152, November 1982.

[11] Patrick E. O'Neil. The escrow transaction method. *ACM Transactions on Database Systems*, 11(4):405–430, December 1986.

[12] John A. Richards. *Remote Sensing Digital Image Analysis*. Springer-Verlag, 1986.

[13] Carl Staelin and Hector Garcia-Molina. Clustering active disk data to improve disk performance. Technical Report CS-TR-283-90, Department of Computer Science, Princeton University, Princeton, NJ, September 1990.