01 Jan 1987

A Dynamic Caching Algorithm based on C. C. Chang's Ordered Minimal Perfect Hashing Scheme

Thomas J. Sager

A DYNAMIC CACHING ALGORITHM BASED ON C. C. CHANG'S ORDERED MINIMAL PERFECT HASHING SCHEME

Thomas J. Sager

CSc-87-2

Department of Computer Science
University of Missouri-Rolla
Rolla, MO 65401 (314) 341-4491

A DYNAMIC CACHING ALGORITHM BASED ON C. C. CHANG'S ORDERED MINIMAL PERFECT HASHING SCHEME

by Thomas J. Sager

CR Categories and Subject Descriptors:

E.2 [Data Storage Representation]: hash-table representation.
H3.3 [Information Storage and Retrieval]: retrieval models, search process, selection process.
B3.2 [Design Styles]: cache memories.

General Terms: Algorithms, Theory, Performance.

Additional Key Words and Phrases:

Searching, hashing, caching, minimal perfect hashing, Chinese remainder theorem.

Abstract:

An algorithm for storage and retrieval in a small dynamic cache is presented. This algorithm is based on C.C. Chang's ordered minimal perfect hashing scheme. Our algorithm has the property that it divides the set of objects that could occupy the cache into an arbitrary number of equivalence classes. The only restriction on which objects can occupy the cache are that no two objects in the same equivalence class can be in the cache at the same time. Retrieval from the cache is performed in constant time. Update of the cache is performed in logarithmic time.

1. Introduction.

We define a cache to be a structure for storage and retrieval that is small relative to the set of objects of interest. A dynamic cache, is a cache whose contents vary with time. Typically, we desire that access to a dynamic cache be faster than access to objects not in the cache and that replacement of one item in the cache by another be reasonably fast. Two of the traditional methods for dynamic caching are: 1. Associative memories in which the object to be accessed is simultaneously compared with all items in the cache. With this method there are no restrictions on which set of objects may occupy the cache at any given time. 2. A hashing scheme in which the set of objects, W, is divided into n equivalence classes where n is the maximum number of objects that the cache can hold. Each equivalence class is then associated with a particular position in the cache. With this method, no two objects from the same equivalence class may occupy the cache at the same time. In both of these methods, collisions do not occur, so rehash is unnecessary.

In this paper we present the Chinese remainder caching algorithm (CRCache). This technique lies somewhere between the above two methods. In the CRCache algorithm, W is divided up into m equivalence classes where $m \geq n$. The only restriction on which sets can occupy the cache is that no two items from the same equivalence class can occupy the cache at the same time. This method although not as fast as the above mentioned two methods, does not require parallel hardware like associative memories, and does not restrict cache occupancy to the extent of the

traditional hashing technique. Although retrieval from the cache is performed in constant time, updating of the cache is performed in logarithmic time.

The CRCache algorithm is based on C.C. Chang's ordered minimal perfect hashing scheme [1,2]. It would seem to be most applicable to applications such as caching a small number of identifiers which are frequently used locally while parsing a computer program.

Section 2 contains a brief review of Chang's ordered minimal perfect hashing scheme. The CRCache algorithm is presented in section 3 and analyzed in section 4. Section 5 contains some concluding remarks.

2. Chang's ordered minimal perfect hashing scheme.

A minimal perfect hashing function (MPHF) is a bijection from a set of n objects to the set, {0..n-1}. Chang's ordered minimal perfect hashing scheme is based on the Chinese remainder theorem which states that:

given 2n non-negative integers m_1, m_2, ..., m_n and r_1, r_2, ..., r_n such that $\forall$ i : $1 \leq i \leq n$, $r_i < m_i$ and $\forall$ i,j : $1 \leq i < j \leq n$, $gcd(m_i, m_j) = 1$,

then $\exists!$ C : $\forall$ i : $1 \leq i \leq n$, $r_i = C \bmod m_i$ and $C < (\prod m_i,\ i := 1 \text{ to } n)$.

In his original presentation [1], Chang showed that given a set, $W = \{w_0, w_1, \ldots w_{n-1}\}$ of objects and a prime number function, p, mapping W into n distinct prime numbers, there is a very elegant and efficient way of calculating C : C <

$(\prod p(w_i),\ i := 0 \text{ to } n-1)$ and $\forall\ i \mid 0 \leq i \leq n-1,\ i = C \bmod p(w_i)$. Thus, $C \bmod p(w_i)$ is a MPHF for W. Chang's work was harshly criticized by Pritchard [4] and Jin [3] for being impractical. However, in a later presentation [2], Chang showed that these criticisms were premature by presenting a letter oriented variant of his algorithm which is quite practical for generating minimal perfect hash functions. The minimal perfect hashing functions generated by this variant are extremely elegant although probably not as fast as those generated by the mincycle algorithm [5] or applicable to sets as large as those to which the mincycle algorithm can be applied.

Let W be the set of n words for which we desire a MPHF. In Chang's letter ordered variant, two character positions, i and j are chosen such that $\forall$ w and w' in W with w<>w', either w[i]<>w'[i] or w[j]<>w'[j], where w[i] represents the ith letter of w. Three arrays, C, d and p, indexed by letters are used. The objects in W are given appropriate subscripts w_1, w_2, ..., w_n such that all words in W with the same letter in position j have contiguous subscripts. The d array is initialized so that for each letter x, d[x] is the smallest integer such that $w_{d[x]}[j] = x$. The array, p, is initialized to contain appropriate prime numbers for every letter that occupies position i for some word in W. Now for each letter, x, which occupies position j for some word in W, we initialize C[x] so that $\forall$ k | $w_k[j]=x$, $k-d[w_k[j]] = C[w_k[j]] \bmod p(w_k[i])$. The function, C[w[j]] mod p[w[i]] + d[w[j]] is therefore a MPHF. The C array is computed by Chang's original algorithm. The heart of Chang's method for computing C is a very elegant algorithm which, given two relatively prime

positive integers p and q, computes b such that b*p mod q = 1 and the observation that:

$C = (\Sigma b_i * M * r_i / m_i,\ i := 1 \text{ to } n) \bmod M$ where

$M = (\Pi m_j,\ j := 1 \text{ to } n)$,

C, the m_i's and r_i's are as in the Chinese remainder theorem and

$\forall i : 1 \leq i \leq n,\ (b_i * M / m_i) \bmod m_i = 1$.

Henceforth, we will call the algorithm for computing b, EQUIV1. EQUIV1 can be shown to have worst case time complexity logarithmic in min(p,q). The reader is refered to Chang's article [1] for the details of EQUIV1.

3. The Chinese Remainder Caching Algorithm.

The cache we wish to create will be represented by an array, TABLE, of n objects indexed by 0..n-1. We replace Chang's character positions, i and j, by two pseudo-random functions, h1 and h2 on our domain of objects. The range of h1 is a small set, R1, of pairwise relatively prime positive integers, such that $\forall$ x in R1, $x \geq n$ and $M = (\Pi x,\ x \text{ in } R1) \leq$ the largest positive integer conveniently representable by the underlying computer system. Naturally the choice of ranges for h1 is highly hardware dependent. R2 = range(h2) is arbitrary, but for our purposes, we may consider that R2 = {0..r-1} where r is a small power of 2.

Now let C and D be arrays of integers indexed by the set R2. Let $D[x] = (\Pi h1(w),\ w \text{ in TABLE and } h2(w)=x)$ and $\forall$ w in TABLE, let C[h2(w)] mod h1(w) = the index of the position in TABLE which w occupies. $\forall$ x in R2, let C[x] < D[x]. Finally, if $\forall$ w in TABLE, h2(w) <> x then let D[x]=1 and C[x]=0.

The Chinese remainder theorem assures us that, provided that $\forall$ w and w' in TABLE with w<>w', either h1(w)<>h1(w') or h2(w)<>h2(w'), the array C is well-defined. Our choice of h1 assures us that all numbers in the C and D arrays can be represented conveniently on our computer system. TABLE, C and D can be initialized to some precomputed values or else TABLE can be initialized with null values such as strings of blanks, C with 0's and D with 1's. Any reasonable replacement algorithm such as round robin or least recently used (LRU) may be employed. Henceforth, when we say optimal replacement object, we mean according to whichever replacement algorithm is being used. Algorithm CRCACHE for cache retrieval and update is given on the following page.

Example 1: Assume the underlying hardware is an IBM 370 and we desire a table of size 16. Since the IBM 370 uses 32 bit two's complement integers, the product of all the elements in range(h1) must be $\leq 2^{31}-1$ for convenient representation on the underlying hardware. Therefore, let:

R1 = {16, 17, 19, 21, 23, 25, 29},

h1(w) = g[ord(first(w)) mod 7] where g[0]=16, g[1]=17, g[2]=1
g[3]=21, g[4]=23, g[5]=25 and g[6]=29 and

h2(w) = (Σ w[i], 1 < i $\leq$ length(w)) mod 16.

Suppose we desire a table of size 32. If we used a machine with 64 bit two's complement numbers or we were willing to pay the price of using 64 bit integers on an IBM 370 without supporting hardware then we could let: R1 = {32, 33, 35, 37, 41, 43, 47, 53, 59, 61, 67} and redefine h1 and h2 appropriately.

```
Algorithm CRCACHE(w: object);

  global
     object:          type;
     n, r:            integer;
     TABLE:           array [0..n-1] of object;
     C, D:            array [0..r-1] of integer;

  local
     index, a1, a2, x, b1, b2:  integer;
     h1, h2:          function(w: object): integer
          { any fast pseudo-random functions };
     EQUIV1:          function(p,q: integer): integer
          { assert gcd(p,q) = 1;   p>1;   q>1;       }
          { returns b | 1 = b*p mod q;               }
          { see Chang [1] for details of function };

  begin
     a1 := h1(w);
     a2 := h2(w);
     index := C[a2] mod a1;
     if index < n and TABLE[index] = w then
       {found};
     elsif index < n and h1(TABLE[index])=a1 and
           h2(TABLE[index])=a2 then
       TABLE[index] := w
             { in this case we may not necessarily }
             { be replacing the optimal object     }
     else
       index := index of optimal replacement object in TABLE;
       x     := h2(TABLE[index]);     {remove object from TABLE}
       D[x]  := D[x] / h1(TABLE[index]);
       C[x]  := C[x] mod D[x];
       TABLE[index] := w;                    {place w in TABLE}
       if D[a2] = 1 then                     {update C and D}
         C[a2] := index;
         D[a2] := a1;
       else
         b1    := EQUIV1(a1, D[a2]);        {b1*a1 mod D[a2] = 1}
         b2    := (1-b1*a1) div D[a2];      {b2*D[a2] mod a1 = 1}
         x     := a1 * D[a2];
         C[a2] := (b1 * a1 * C[a2] + b2 * D[a2] * index) mod x;
         D[a2] := x;
       endif
     endif
  end CRCACHE;
```

Example 2: Suppose we wish to use an LRU replacement scheme and we let:

n = 4;

$$h1(w) = \begin{cases} 5 & \text{if } ord(first(w)) \bmod 3 = 0 \\ 7 & \text{if } ord(first(w)) \bmod 3 = 1 \\ 4 & \text{if } ord(first(w)) \bmod 3 = 2; \end{cases}$$

h2(w) = ord(last(w)) mod 2;

	index		0	1	2	3
Suppose initally our system looks like this where the LRU row gives order of least recent use.	TABLE	=	AA	AB	BA	CB
	C	=	17	12		
	D	=	28	20		
	LRU	=	4	3	2	1
Suppose the next word is BA since BA is already in the table only LRU is changed	LRU	=	3	2	4	1
Suppose the next word is CA. CA is placed at least recently used index with the result that:	TABLE	=	AA	AB	BA	CA
	C	=	1	52		
	D	=	4	140		
	LRU	=	2	1	3	4
Suppose the next word is DA. Since h1(DA)=h1(AA) and h2(DA)=h2(AA), DA must replace AA instead of the least recently used AB. This yields:	TABLE	=	DA	AB	BA	CA
	C	=	1	52		
	D	=	4	140		
	LRU	=	4	1	2	3

4. An Analysis of the Chinese Remainder Caching Algorithm.

Each invocation of algorithm CRCACHE can be considered to belong to one of three cases.

case 1: The object, w, to be looked up is in TABLE. In this case the index is found in constant time.

case 2: The object, w, to be looked up is not in TABLE and neither is any object w' such that h1(w)=h1(w') and h2(w)=h2(w'). In this case w replaces the optimal replacement object (according to some algorithm). The worst case time necessary to update the C and D arrays is proportional to the log of the largest number in R1.

case 3: The object, w, to be looked up is not in TABLE, but an object, w', such that h1(w)=h1(w') and h2(w)=h2(w') is in TABLE. In this case the index at which to place w is again found in constant time and no updating of the arrays C and D is necessary. However, the object replaced is not necessarily the optimal object. Assuming that the functions h1 and h2 are truly random, then this happens with the probability (n-1)/(r*|R1|) when w is not in TABLE.

In terms of space, CRCACHE uses glabal storage for the arrays C, D and TABLE and for the replacement algorithm. Only the arrays C and D need be considered unique to CRCACHE, since TABLE and a replacement algorithm would be necessary to any algorithm of this type. The C and D arrays each take up space for r integers. This value, therefore, represents a tradeoff between space and the probability of occurance of case 3 above.

It should be noted that the functions h1 and h2 divide W into r * |R1| equivalence classes. The only restriction on the objects in TABLE is that no two objects from the same equivalence class can reside in TABLE at the same time.

5. Conclusion

In the previous sections we have presented the CRCache algorithm for storage and retrieval of objects. This algorithm is based on C.C. Chang's ordered minimal perfect hashing scheme. CRCache has the interesting quality that it divides the domain of objects into an arbitrary number of equivalence classes such that the only restriction on the contents of the cache are that no two objects from the same equivalence class may occupy the cache at the same time.

References

[1] Chang, C.C. The study on an ordered minimal perfect hashing scheme. Commun. ACM, 27, 4, (Apr. 1984), 384-387.

[2] Chang, C.C. A letter oriented minimal perfect hashing scheme. The Computer Journal, 29, 3, (June. 1986), 277-281.

[3] Jin, Suigen. Technical Note. Commun. ACM, 28, 10, (Oct. 1985), 1087.

[4] Pritchard, Paul. Technical Note. Commun. ACM, 27, 11, (Nov. 1984), 1156-1157.

[5] Sager, T.J. A polynomial time generator for minimal perfect hash functions. Commun. ACM, 28, 5, (May 1985) 523-532.