Analyzing Hash Influence on Branch Prediction Accuracy

Mojtaba Shakeri

mshakeri@ce.aut.ac.ir

Mohammad K. Akbari

akbari@ce.aut.ac.ir

Bahman Javadi

javadi@ce.aut.ac.ir

Department of Computer Engineering and Information Technology
Amirkabir University of Technology

Abstract: *Branch prediction is important in high-performance microprocessor design. Among branch prediction mechanisms, dynamic branch predictors are the best because they can deliver accurate branch prediction regardless of input changes or the program behavior. However in all existing dynamic branch predictors, the interference effects due to aliasing in the prediction tables are the most important causes of branch mispredictions.*

In this paper some popular hash functions used in different computer applications are proposed and their effects on dynamic branch predictors are analyzed. We investigate hash influence on branch prediction accuracy in both analytical and experimental ways. Our experimental results suggest that hash can be incrementally effective on branch prediction accuracy and this increase is dependent on branch behavior.

Keywords- Branch prediction, hash, uniformity, branch behavior.

1. Introduction

Branch prediction is important in high-performance microprocessor design. There are two trends that are increasing the importance of branch prediction further: Processors are getting wider and pipelines are getting deeper. These trends make branch prediction critical, because they increase the relative cost of branch mispredictions [1]. It is argued that by the year 2010 branch prediction will become the most limiting factor in processor performance, surpassing even the limitation of memory system [2].

Among different branch prediction mechanisms, dynamic branch predictors are more popular because they can deliver accurate branch prediction without changes to the instruction set or pre-existing binaries [3]. But the main problem that reduces the prediction rate in all existing dynamic prediction schemes (Gshare [4], The Agree Predictor [5], The Bi-Mode Predictor [3], The Skew Branch Predictor [6], The Filter Mechanism [7], and The YAGS Branch Predictor [8]) is aliasing between two indices (an index is typically formed from history and address bits) that map to the same entry in the prediction table [8]. One way to reduce aliasing is to improve the indexing method. That is, in addition to using the branch address as its argument, it takes other arguments representing current program behavior along with randomization to reduce aliasing. In this paper we propose *hash* to be the solution.

Hash normally consists of three parts: a hash table, key(s), and a hash function. A hash table is an effective data structure for implementing the dictionary operations of INSERT, SEARCH, and DELETE [9]. In this paper, we are interested in SEARCH[1]: given a branch address as our key, we *search* the hash table to predict its direction[2]. The point here is instead of using the key as an array index

---

[1] We can assume INSERT, as the operation updating a prediction-bit counter after a branch is resolved.

[2] In this paper, we are interested in predicting the direction of branches not their targets.

directly, the array index is computed from the key. This is what a hash function is supposed to do. Of course due to its critical path, search for a branch direction in the hash table should not exceed $o(1)$ time.

Due to hardware difficulties and delay concerns, practical implementation of almost all of dynamic branch predictors [3,4,5,6,7,8] applied the simple linear hashing of modulo-2 addition (XOR function) using branch address as their primary input (hash-key) and global/local history as their secondary one. It should be added that Neural Branch Prediction [10], a recent concept in branch prediction, also exploits hashing techniques. It uses the adaptive and flexible behavior of neural networks to predict branch directions [11]. In this technique, table of perceptrons (or neurons) is indexed by address hash. These perceptrons contain array of weights corresponding to history bits [10]. By using a hash method based on multiplying history bits by their respective weights, direction of a given branch is determined. This new technique needs more work to meet current hardware and timing requirements and we do not elaborate on it any more.

In this paper, different models using a particular hash function are analyzed. Section 2, compares a direct access table with a hash table. Section 3, introduces some common hash functions used in different computer applications now targeted at branch prediction. Section 4 does an analysis on access probability of each entry and categorizes them into 4 classes. Section 5 describes our simulation methodology. Section 6 presents our experimental results. And section 7 concludes this study and proposes our future research directions.

2. Direct-Access Table vs. Hash Table

Direct addressing is a simple technique that works well when the universe U of keys is reasonably small [9], i.e. total number of branches executed is comparable with number of static branches in a program. This is our base model (figure1). Successive models presented next are compared with this model.

When a branch is fetched, its key (i.e. its address) is drawn from the universe $U = \{0,1,...,m-1\}$. To obtain the direction,

we use an array or a direct-address table, denoted by $T = [0..m-1]$ in which each position or slot (i.e. branch predicted direction) corresponds to a key in universe U. Now the search operation is defined as [9]:

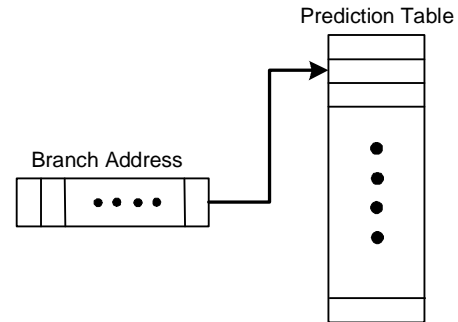*DIRECT_ADDRESS_SEARCH (T,k)*
return T[k]

Figure1: A direct address indexing model.

The difficulty with direct addressing is obvious: if the universe U is large, storing a table T of size $|U|$ may be impractical, or even impossible. Furthermore, the set k of keys actually stored may be so small relative to U that most of the space allocated for T would be wasted [9]. That is, many elements would be mapped to the same slots. In other words, it is not possible to hold all relevant branch history for all active branches at the same time [6] and branch interference would be considerable if only branch addresses were used for indexing. (Here $|U|$ is number of dynamic branches and $|k|$ is number of static branches).

With direct addressing, a branch direction with address k is found in entry k. While with hashing, its direction is found in entry $h(k)$. That is, we use a hash function h to compute entry (slot) from the address (key) k. Figure 2 shows how hashing is used in our approach. Here h maps the universe U of keys into the slots of a hash table $T = [0..m-1]$:

$$h : U \rightarrow \{0,1,...,m-1\}$$

Prediction Table

Branch Address
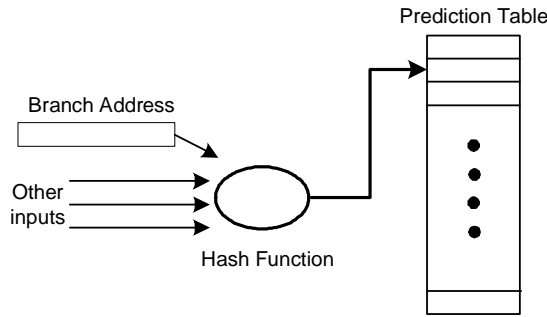
Other
inputs

Hash Function

Figure2: How hashing can be used for branch
prediction.

Similar to direct address indexing, we still
may have the problem of aliasing: two keys
may be mapped to the same slot. This
situation is also called a collision [9]. The
ideal solution would be to avoid collisions
altogether. We might try to achieve this goal
by choosing a suitable hash function *h*. One
idea is to make *h* appear to be random, thus
avoiding collisions or at least minimizing
their number. The very term "to hash"
evoking images of random mixing and
chopping captures the spirit of this approach
[9]. Next section discusses hash functions that
somehow consider randomization in their
mappings.

3. Hash Functions

In this section some well-known hash
functions are introduced, but before that, let's
know what makes a good hash function. A
good hash function satisfies (approximately)
the assumption of simple uniform hashing;
each key is equally likely to hash to any of
the *m* slots, independently of where any other
key has hashed to [9]. Unfortunately it is
typically not possible to check this condition,
since one rarely knows the probability
distribution according to which the keys are
drawn and the keys may not be drawn
independently. This problem is particularly
obvious for branch prediction, since each
branch is dependent upon previous executions
of itself or other branches. We will talk about
uniformity more in section 4.

3.1 The division method

In the division method for creating hash
functions, we map a key into one of *m* slots
by taking the remainder of *k* divided by *m*.
Therefore, the hash function is [9]:

$$h(k) = k \mod m$$

Where in our design, *k* is the branch
address and *m* is the length of the prediction
table. Should *m* be a power of two, then this
model simply gets our base model defined in
section 2 (e.g. if $m = 2^p$ then *h(k)* is just the *p*
lowest-order bits of *k*). Unless it is known
that all low-order *p*-bit patterns are equally
likely, it is better to make the hash-function
depend on all the bits of the key [9].

Ignoring hardware difficulties we would
encounter while implementing this method,
we choose a prime not too close to an exact
power of two. It is often a good choice for *m*
[9]. *(It should be reminded that in this paper
we are interested in realizing the effectiveness
of hashing on branch prediction accuracy).*

3.2 The multiplication method

The multiplication method for creating
hash functions operates in two steps. First, we
multiply the key *k* by a constant *A* in the
range $0 < A < 1$ and extract the fractional part
of *kA*. Then we multiply this value by *m* and
take the floor of the result. In short, the hash
function is [9]:

$$h(k) = \lfloor m(kA \mod 1) \rfloor$$

Where "$kA \mod 1$" means the fractional
part of *kA*, that is, $kA - \lfloor kA \rfloor$. An advantage
of the multiplication method is that the value
of *m* is not critical. So we choose it to be a
power of two ($m = 2^p$ for some integer *p*) to
be hopeful that it would be easily
implemental in hardware!

Suppose that the word size of our
architecture is *w*-bits (here $w = 32$) and that *k*
fits into a single word. We restrict *A* to be a
fraction of the form $s/2^w$, where *s* is an
integer in the range $0 < s < 2^w$. Referring to
figure 3, we first multiply *k* by the *w*-bit
integer $s = A \cdot 2^w$. The result is a 2*w*-bit value
$r_1 2^w + r_0$, where r_1 is the high order word of
the product and r_0 is the low-order word of
the product. The desired *p*-bit hash value
consists of the *p* most significant bits of r_0
(practically $p << m$) [9].

Although this method works with any
value of the constant *A*, it has been shown
that

$$A \approx (\sqrt{5} - 1)\big/ 2 = 0.6180339887...$$

is likely to work reasonably well [9].

For our architecture, where $w = 32$, we have $s = 2654435769$ and $A = s\big/2^{32}$.
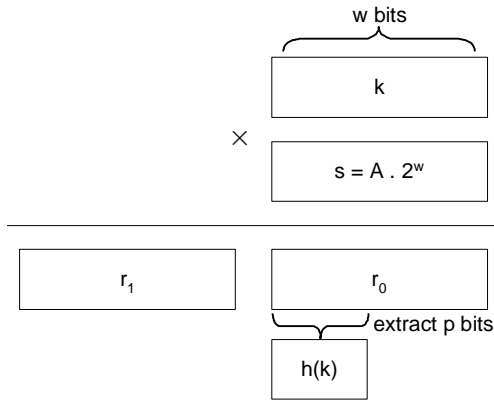

Figure3: The multiplication method of hashing.

3.3 Open addressing

In the three models introduced so far, for a particular branch during a program execution, a particular entry is always used. In other words, if this branch executes several times, its predicted direction is always determined by a specific entry.

It is common notice that a branch behavior might alter during the program execution. Now if a particular entry is only used to predict this branch direction, number of mispredictions occurring could be considerable. Moreover, this entry may be used for other branches of different directions (that is, collision or aliasing). This is one of the drawbacks pointing to address-only branch prediction mechanisms.

Now if we enter another element as an indicator of the present program behavior to our hash function, then we can defeat this shortcoming to great extent. This is the idea behind open addressing. Three techniques are commonly used in open addressing: linear probing, quadratic probing, and double hashing [9] that due to our application, only two of them are addressed here.

3.3.1 Linear probing

Given an ordinary hash function:

$$h' : U \rightarrow \{0,1,...,m-1\}$$

as an auxiliary hash function, the method of linear probing uses the hash function [9]:

$$h(k,i) = (h'(k) + i) \mod m$$

Now if consider k as our branch address, $h'(k)$ as the hash function of our base model (i.e. $h'(k) = k$), i as the global branch history register, and '+' as the modulo–2 addition, then we have the famous g-share branch prediction scheme [4].

In our simulation (discussed in section 5), we take i as:

1. Dynamic branch counter (It is indicative of current number of branches executed dynamically in a program).

2. Branch global history register.

3.3.2 Double Hashing

Double hashing is one of the best methods available for open addressing. Because the patterns produced have many of the characteristics of randomly chosen patterns. Double hashing uses a hash function of the form:

$$h(k,i) = (h'(k) + i \cdot h''(k)) \mod m$$

where h' and h'' are auxiliary hash functions.

The value $h''(k)$ must be relatively prime to the hash-table size m for the entire hash table to be searched. A convenient way to ensure this condition is to let m be a power of two and to design h'' so that it always produces an odd number (e.g. we can OR the LSB bit by one). Another way is to let m be prime and to design h'' so that it always returns a positive integer less than m. For example, we could choose m prime and let:

$$h'(k) = k \mod m,$$
$$h''(k) = 1 + (k \mod m'),$$

where m' is chosen to be slightly less than m (say, $m - 1$).

4. Analysis

In this section, we investigate our approach analytically before we do the simulation.

Given a hash table T with m slots (entries) that is accessed n times during a program execution where $n >> m$, we define the load factor α for T as n/m, that is, the average number of accesses to each slot.

The average performance of hashing depends on how well the hash function h distributes the set of keys to be stored among the m slots, on the average. Here we assume that any given element is equally likely to hash into any of the m slots, independently of where any other element has hashed to. We call this the assumption of simple uniform hashing [9].

For $j = 0,1,...,m-1$, let us denote the number of accesses to $slot_j$ by n_j, so that

$$n = n_0 + n_1 + ... + n_{m-1},$$

and the average value of n_j is

$$E[n_j] = \alpha = {n}/{m}.$$

We also define p_j as $slot_j$ access probability, that is, $p_j = n_j/n$ and the average value of p_j is

$$q = E[p_j] = {E[n_j]}/{n} = {{n}/{m}}/{n} = {1}/{m}$$

Now we use q as our crucial item to classify the prediction table entries into 4 categories ranging from $Category_0$ to $Category_3$:

- $Category_0$: Those entries that remain untouched (useless), during a program execution, belong to this category.

- $Category_1$: This category contains those entries whose access probability ranges from *0* to *q*. In fact, the more entries belong to this category, the more effective the indexing method is.

- $Category_2$: This category consists of those entries whose access probability ranges from *q* to $qlog_2{}^m$. The reason why we have put the parameter $log_2{}^m$ in our classification is to consider the effect of the number of the prediction table entries (i.e. *m*)

in our simulation. In other words, the region of each category is directly related to the number of the prediction table entries.

- $Category_3$: This category contains those entries whose access probability is more than $qlog_2{}^m$.

Now we define the hash functions used in our simulation:

- $func_0 : h_0(k) = k,$
  in which k is the branch address. This is our base model in which the branch address is *directly* used for indexing.

- $func_1 : h_1(k) = k \mod m,$
  where k is the branch address and m is a prime standing for the prediction table size. This hash function is based on the *division* method.

- $func_2 : h_2(k) = \lfloor m(kA \mod 1) \rfloor,$
  in which k is the branch address, m represents the prediction table size and it is a power of two ($m = 2^p$ for some integer p), and A is a constant equal to $(\sqrt{5} - 1)/2$.

- $func_3 : h_3(k,i) = (h'(k) + i) \mod m,$
  where k is the branch address, $h'(k)$ is equal to $h_0(k)$, m represents the prediction table size and it is a power of two ($m = 2^p$ for some integer p), i is the dynamic branch counter, and '+' is the modulo–10 addition.

- $func_4 : h_4(k,i) = (h'(k) + i) \mod m,$
  where k is the branch address, $h'(k)$ is equal to $h_0(k)$, m represents the prediction table size and it is a power of two ($m = 2^p$ for some integer p), i is the global history register, and '+' is the modulo–10 addition.

- $func_5 :$
  $h_5(k,i) = (h'(k) + i \cdot h''(k)) \mod m,$
  where k is the branch address, $h'(k)$ is equal to $h_1(k)$, $h''(k)$ is equal to ($h_0(k)$ mod $m - 1$), m is a prime indicating the prediction table size, i is the dynamic branch counter, and '+' is the modulo–10 addition.

- $func_6$:

  $h_6(k,i) = (h'(k) + i \cdot h''(k)) \mod m$, where k is the branch address, $h'(k)$ is equal to $h_1(k)$, $h''(k)$ is equal to ($h_0(k)$ mod $m - 1$), m is a prime indicating the prediction table size, i is the global history register, and '+' is the modulo–10 addition.

- $func_7$:

  $h_7(k,i) = (h'(k) + i \cdot h''(k)) \mod m$, where k is the branch address, $h'(k)$ is equal to $h_2(k)$, $h''(k)$ is equal to $h_0(k)/1$, m represents the prediction table size and it is a power of two ($m = 2^p$ for some integer p), i is the dynamic branch counter, and '+' is the modulo–10 addition.

- $func_8$:

  $h_8(k,i) = (h'(k) + i \cdot h''(k)) \mod m$, where k is the branch address, $h'(k)$ is equal to $h_2(k)$, $h''(k)$ is equal to $h_0(k)/1$, m represents the prediction table size and it is a power of two ($m = 2^p$ for some integer p), i is the global history register, and '+' is the modulo–10 addition.

5. Simulation Methodology

Performance of each hash function proposed in section 4 was measured by instruction-driven simulations performed on the SPECint2000 benchmarks [12]. Five SPECint2000 benchmarks were used for our evaluation: crafty, gap, gcc, mcf, and twolf. (SPECfp2000 benchmarks are not considered in our simulation, since they present high prediction accuracy even in case of using poor branch prediction mechanisms). We used Simplescalar's *bpred* simulator (modified to fulfill our requirements) [13] as our simulation tool. The execution statistics of these benchmarks are listed in table 1. Benchmarks were run by applying train inputs.

As mentioned before, the word size of our architecture is 32 bits wide. For hash functions h_3, h_5, and h_7 we allocated a 64-bit register to represent the dynamic branch counter and for hash functions h_4, h_6, and h_8 we used 12 bits for global history.

Table1: Execution statistics of SPECint2000 benchmarks

Benchmark	Description	#Simulated Instructions	Instruction Per Branch	Cond. Branch (%)	Uncond. Branch (%)
crafty	Chess program	27,216,357,356	8.6547	10.3	1.2
gap	Computational group theory	9,518,044,317	6.927	13.05	0
gcc	GNU C compiler	5,117,147,311	6.775	13.25	1.2
mcf	Minimum cost network flow solver	9,168,13,14,88	4.4875	19.9	2.3
twolf	Place & route simulator	13,199,966,132	7.9345	11.8	0.8

6. Experimental Results

For the benchmarks listed in table 1, gcc and mcf, while having maximum dynamic conditional branches relative to the other benchmarks, behaved quite differently. So we preferred to compare the results of these two benchmarks with each other. Figures 4 and 5 present the simulation results for these two benchmarks respectively.

For gcc, the division (i.e. h_1) and multiplication (i.e. h_2) methods improved the prediction accuracy by 0.5 percent relative to the direct addressing method (i.e. h_0) on the average. However, for hash functions h_4, h_6, and h_8, we have degradation in the prediction accuracy. Although this degradation decreases as PHT size increases, there is about 5 percent increase in the misprediction rate on the average.

For mcf, the division (i.e. h_1) and multiplication (i.e. h_2) methods made no improvement on the prediction accuracy relative to the direct addressing method (i.e. h_0). Nevertheless, for hash functions h_4, h_6, and h_8, about 2 percent improvement on the prediction accuracy was obtained on the average.

To explain this controversy we should refer to the distribution and behavior of the branch instructions executed in these two benchmarks. If we compare h_0 uniformity chart of gcc with that of mcf, we can notice the difference. As PHT size increases, the number of entries remained untouched dramatically increases for mcf, whereas for gcc this increase is negligible. In addition, hash functions h_4, h_6, and h_8, all use the global history as their second argument (that is, other than the branch address). It seems that for mcf, the branches are highly correlated with each other, while for gcc this correlation is less. Furthermore, for mcf, hash functions h_4, h_6, and h_8, utilized almost all entries of PHT fairly uniformly relative to h_0. While for gcc, for all hash functions, good access uniformity already exists.

For both benchmarks, the misprediction rates (not shown here) for hash functions h_3, h_5, and h_7 relative to h_0 is much too considerable (about 25 to 30 percent increase in the misprediction rate relative to the base

model). The similarity between these hash functions is the use of the dynamic branch counter as their second argument. Although using this parameter provides best uniformity (i.e. all PHT entries, with no exception, belong to $Category_1$), the generated result contradicts our inference. The reason is obvious: the dynamic branch counter cannot represent dynamic behavior of a program. So in many cases there may branches with opposite directions that are hashed to the same entry thus increasing *destructive* [5,7] interference. Now if we could have a dynamic selection mechanism to dynamically classify branches into mostly taken and untaken ones (as what is used in the Bi-Mode Predictor [3]), then we could claim that using the dynamic branch counter would decrease the branch misprediction rate due to the uniformity it yields.

7. Future Work & Conclusion

In this paper, we analyzed the performance of different hash methods mostly used in applications other than computer architecture. For simplicity and ease of understanding, we proposed a simple bimodal branch predictor and ignored hardware difficulties (e.g. delay, area) that could be involved.

We developed our approach by presenting 9 hash functions (including the direct address indexing method). We also classified the prediction table entries into 4 categories according to their access probability to analytically investigate our work.

The experimental results were dependent upon the benchmarks used for the simulation. That is, for different benchmarks different results were obtained. This difference was mainly significant for gcc and mcf benchmarks whose branch behavior differs from each other.

Using the dynamic branch counter as the second argument for hashing (beside the branch address) dramatically degraded the prediction accuracy despite yielding best access uniformity. The reason of this degradation is the numerous destructive interferences between oppositely-biased branches for most prediction table entries.
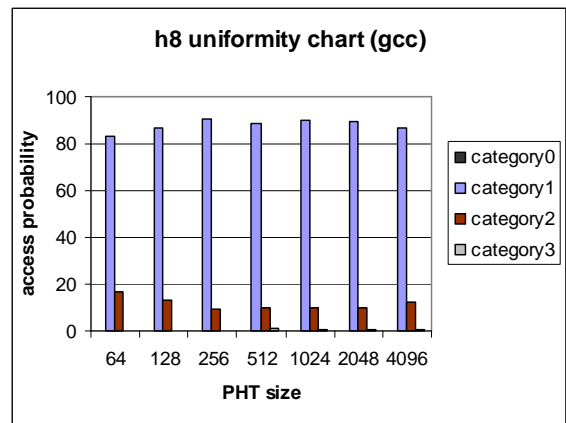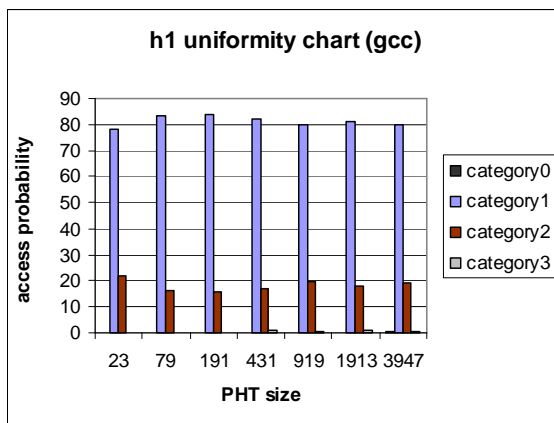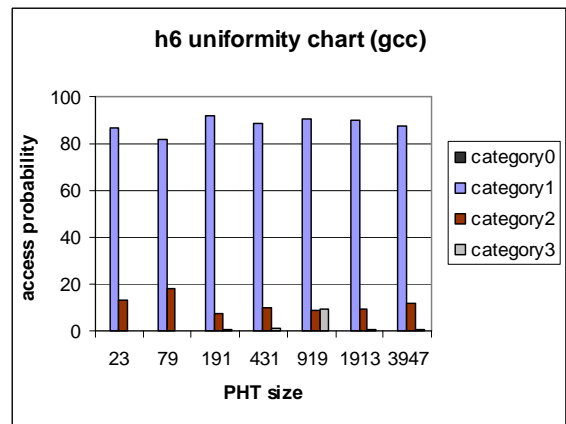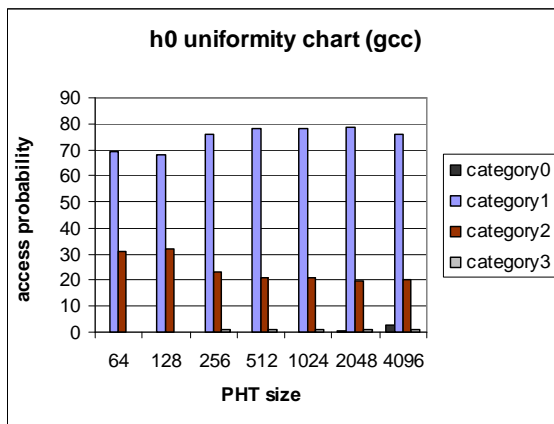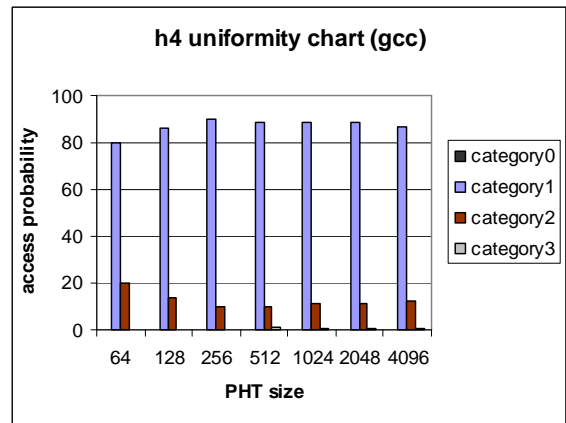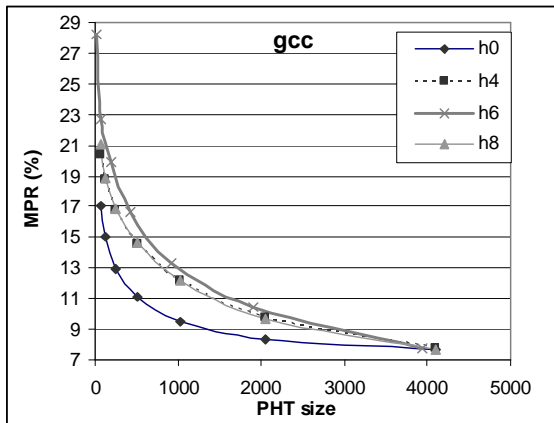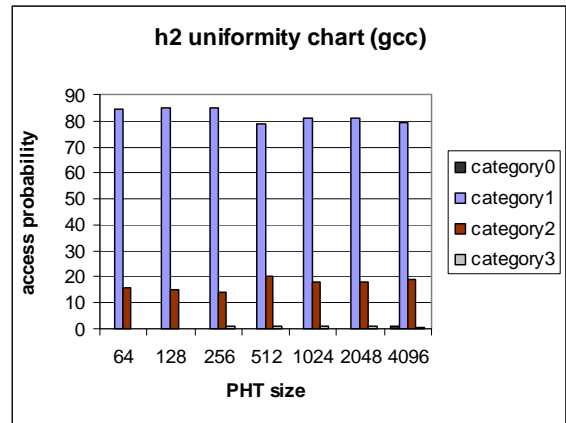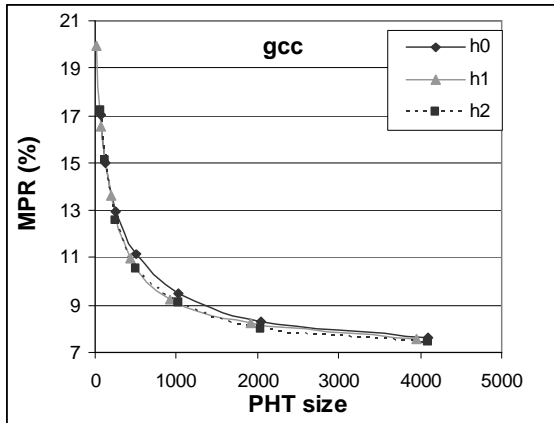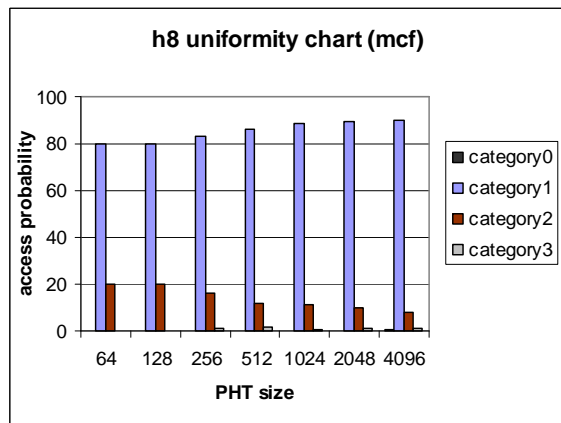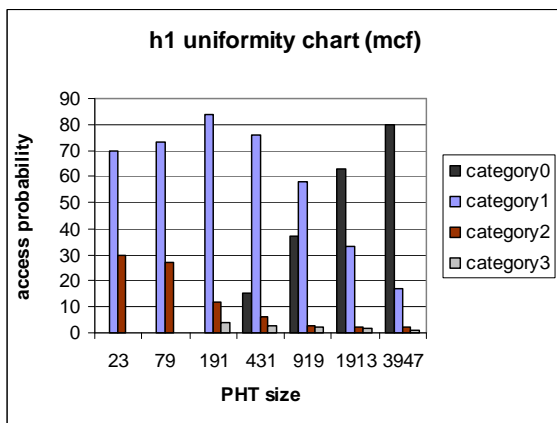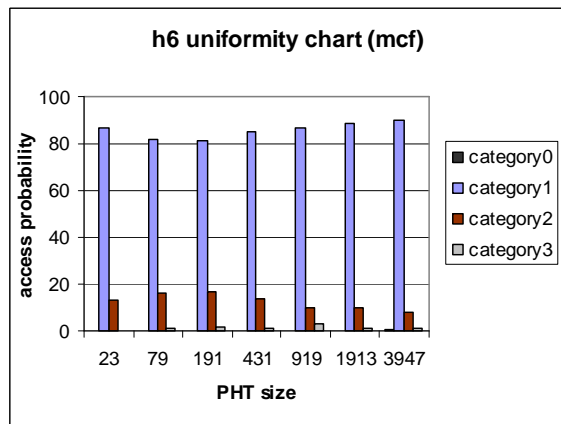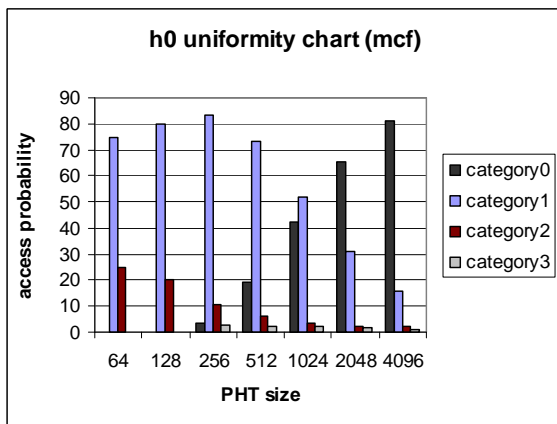
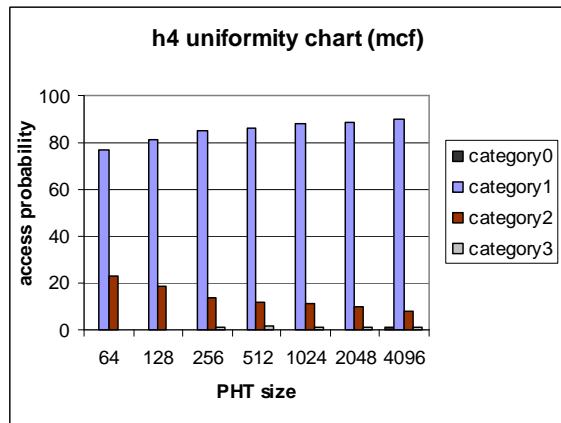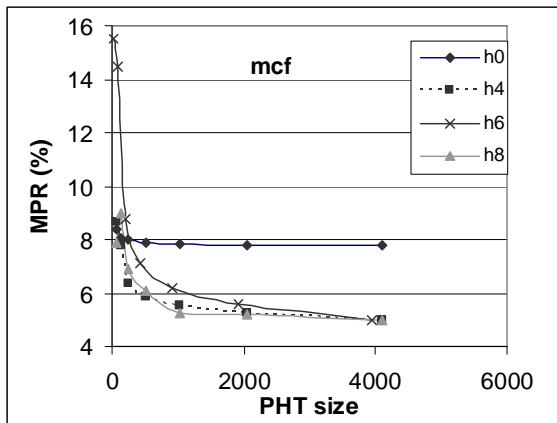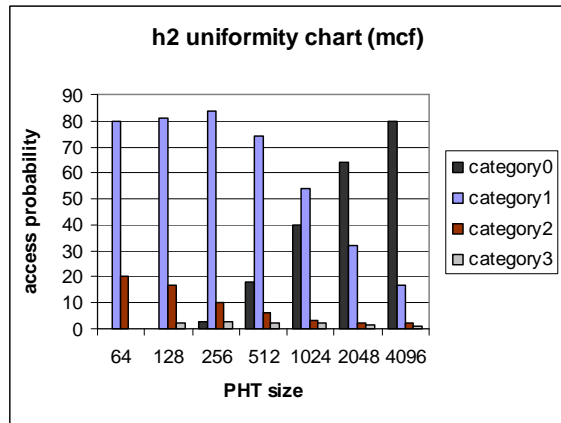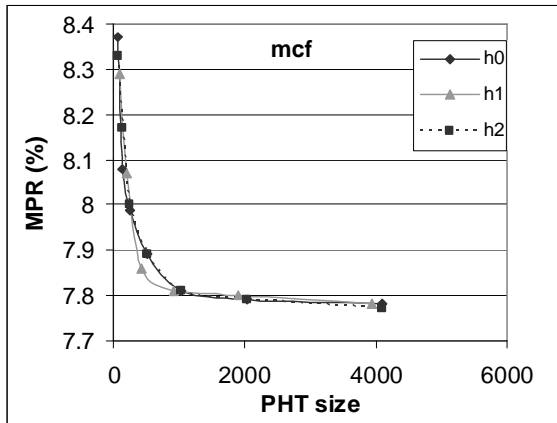Figure4: The simulation results for benchmark gcc.

Figure5: The simulation results for benchmark mcf

The authors believe that by applying a dynamic selection mechanism to dynamically classify branches into heavily taken and heavily untaken ones, we can best make use of access uniformity. The important point here is how to design this dynamic selection mechanism that can work well for all benchmarks with different branch behavior.

Furthermore, by bringing more parameters representing the current behavior of a program into our proposed hash functions, it is possible to strengthen hash influence on branch prediction accuracy. We are currently investigating these issues.

Acknowledgement

References

[1] M. Evers, and T.-Y., "Understanding branches and designing branch predictors for high-performance microprocessors," invited paper in *Proc. IEEE*, Vol. 89, No. 11, pp. 1610-1620, 2001.

[2] A. N. Eden, J. Ringenberg, S. Sparrow, and T. N. Mudge, "Hybrid myths in branch prediction," in *Proc. the 7th Int. Conf. on Information Systems Analysis and Synthesis (ISAS 2001)*, 2001.

[3] C.-C. Lee, I.-C. K. Chen, and T. N. Mudge, "The bi-mode branch predictor," in *30th ACM/IEEE Int. Symp. Microarchitecture*, 1997.

[4] S. McFarling, "Combining branch predictors," Digital Equipment Corporation, WRL Tech. Note TN-36, 1993.

[5] E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt, "The agree predictor: A mechanism for reducing negative branch history interference," in *Proc. 24th Annu. Int. Symp. Computer Architecture*, 1997.

[6] P. Michaud, A. Seznec, and R. Uhlig, "Trading conflict and capacity aliasing in conditional branch predictors," in *24th Annu. Int. Symp. Computer Architecture*, 1997.

[7] P.-Y. Chang, M. Evers, and Y. N. Patt, "Improving branch prediction accuracy by reducing pattern history table interference," in *Proc. Int. Conf. Parallel Architectures and Compilation Techniques*, 1996.

[8] A. N. Eden and T. N. Mudge, "The YAGS branch predictor," in *Proc. 31st Int. Symp. Computer Architecture*, 1998.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms* (Second Edition), published by MIT Press and McGraw-Hill, 2001.

[10] J. Jenkins and J. M. Phillips, "Feature Added Branch Prediction In Multilevel Pipelines," Technical Report, Dec. 2003.

[11] C. M. Bishop, *Neural Networks for Pattern Recognition*, published by Oxford University Press, 2000.

[12] Standard Performance Evaluation Corp, SPECCPU2000 Benchmarks, http://www.specbench.org

[13] D. Burger, T. Austin, "The simplescalar tool set, version2.0," Technical Report TR 1342, University of Wisconsin, 1997.