Using Prime Numbers for Cache Indexing to Eliminate Conflict Misses *

Mazen Kharbutli, Keith Irwin, Yan Solihin
Dept. of Electrical and Computer Engineering
North Carolina State University
{mmkharbu,kirwin,solihin}@eos.ncsu.edu

Jaejin Lee
School of Computer Science and Engineering
Seoul National University
jlee@cse.snu.ac.kr

Abstract

Using alternative cache indexing/hashing functions is a popular technique to reduce conflict misses by achieving a more uniform cache access distribution across the sets in the cache. Although various alternative hashing functions have been demonstrated to eliminate the worst case conflict behavior, no study has really analyzed the pathological behavior of such hashing functions that often result in performance slowdown. In this paper, we present an in-depth analysis of the pathological behavior of cache hashing functions. Based on the analysis, we propose two new hashing functions: prime modulo *and* prime displacement *that are resistant to pathological behavior and yet are able to eliminate the worst case conflict behavior in the L2 cache. We show that these two schemes can be implemented in fast hardware using a set of narrow add operations, with negligible fragmentation in the L2 cache. We evaluate the schemes on 23 memory intensive applications. For applications that have non-uniform cache accesses, both prime modulo and prime displacement hashing achieve an average speedup of 1.27 compared to traditional hashing, without slowing down any of the 23 benchmarks. We also evaluate using multiple prime displacement hashing functions in conjunction with a skewed associative L2 cache. The skewed associative cache achieves a better average speedup at the cost of some pathological behavior that slows down four applications by up to 7%.*

1. Introduction

Despite the relatively large size and high associativity of the L2 cache, conflict misses are a significant performance bottleneck in many applications. Alternative cache indexing/hashing functions are used to reduce such conflicts by achieving a more uniform access distribution across the L1 cache sets [18, 4, 22, 23], L2 cache sets [19], or the main memory banks [14, 10, 15, 16, 20, 8, 7, 26, 11]. Although various alternative hashing functions have been demonstrated to eliminate the worst case conflict behavior, few studies, if any, have analyzed the pathological behavior of such hashing functions that often result in performance degradation.

This paper presents an in-depth analysis of the pathological behavior of hashing functions and proposes two hashing functions: *prime modulo* and *prime displacement* that are resistant to the pathological behavior and yet are able to eliminate the worst case conflict misses for the L2 cache. The number of cache sets in the prime modulo hashing is a prime number, while the prime displacement hashing adds an offset, equal to a prime number multiplied by the tag bits, to the index bits of an address to obtain a new cache index. The prime modulo hashing has been used in software hash tables [1] and in the Borroughs Scientific Processor [10]. A fast implementation of prime modulo hashing has only been proposed for Mersenne prime numbers [25]. Since Mersenne prime numbers are sparse (i.e., for most n, $2^n - 1$ are not prime), using Mersenne prime numbers significantly restricts the number of cache sets that can be implemented.

We present an implementation that solves the three main drawbacks of prime modulo hashing when it is directly applied to cache indexing. First, we present a fast hardware mechanism that uses a set of narrow add operations in place of a true integer division. Secondly, by applying the prime modulo to the L2 cache, the fragmentation that results from not fully utilizing a power of two number of cache sets becomes negligible. Finally, we show an implementation where the latency of the prime modulo computation can be hidden by performing it in parallel with L1 accesses or by caching the partial computation in the TLB. We show that the prime modulo hashing has properties that make it resistant to the pathological behavior that plagues other alternative hashing functions, while at the same time enable it to eliminate worst case conflict misses.

Although the prime displacement hashing lacks the theoretical superiority of the prime modulo hashing, it can perform just as well in practice when the prime number

*This work was supported in part by North Carolina State University, Seoul National University, the Korean Ministry of Education under the BK21 program, and the Korean Ministry of Science and Technology under the National Research Laboratory program.

is carefully selected. In addition, the prime displacement hashing can easily be used in conjunction with a skewed associative cache that uses multiple hashing functions to further distribute the cache set accesses. In this case, a unique prime number is used for each cache bank.

We use 23 memory intensive applications from various sources, and categorize them into two classes: one with non-uniform cache set accesses and the other with uniform cache set accesses. We found that on the applications with non-uniform accesses, the prime modulo hashing achieves an average speedup of 1.27. It does not slow down any of the 23 applications except in one case by only 2%. The prime displacement hashing achieves almost identical performance to the prime modulo hashing but without slowing down any of the 23 benchmarks. Both methods outperform an XOR based indexing function, which obtains an average speedup of 1.21 on applications with non-uniform cache set accesses.

Using multiple hashing functions, skewed associative caches sometimes are able to eliminate more misses than using a single hashing function. However, we found that the use of a skewed associative cache introduces some pathological behavior that slows down some applications. A skewed associative cache with the XOR-based hashing [18, 4, 19] obtains an average speedup of 1.31 for benchmarks with non-uniform accesses, but slows down four applications by up to 9%. However, when using the prime displacement hashing that we propose in conjunction with a skewed associative cache, the average speedup improves to 1.35, and the worst case slowdown drops to 7%.

The rest of the paper is organized as follows. Section 2 discusses the metrics and ideal properties of a hashing function, which help in understanding the pathological behavior. Section 3 discusses the proposed hashing functions: prime modulo and prime displacement and their implementations. Section 4 describes the evaluation environment while Section 5 discusses the results obtained. Section 6 lists the related work. Finally, Section 7 concludes the paper.

2. Properties of Hashing Functions

In this section, we will describe two metrics that are helpful in understanding pathological behavior of hashing functions (Section 2.1), and two properties that a hashing function must have in order to avoid the pathological behavior (Section 2.2).

2.1. Metrics

The first metric that estimates the degree of the pathological behavior is *balance*, which describes how evenly distributed the addresses are over the cache sets. The other is *concentration*, which measures how evenly the sets are used over small intervals of accesses.

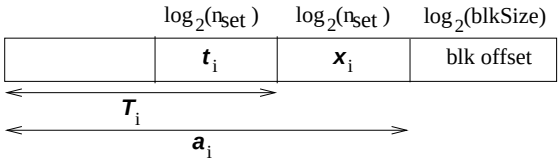


Figure 1: Components of a block address a_i.

Let n_{set} be the number of sets in the cache that is a power of two. This implies that $\log_2 n_{set}$ bits from the address are used to obtain a set index. Let a sequence $< a_1, a_2, ..., a_m >$ of block addresses a_i denote m cache accesses. Suppose $a_i \neq a_j$ for $i \neq j$ where $1 \leq i, j \leq m$, implying that each block address in the sequence is unique. Thus, the address sequence does not have any temporal reuse (we will return to this issue later). Let H be a hashing function that maps each block address a_i to set $H(a_i)$ in the cache. We denote $\log_2 n_{set}$ index bits of a_i with x_i and the first $\log_2 n_{set}$ bits of the tag of a_i with t_i as shown in Figure 1.

Balance describes how evenly distributed the addresses are over the sets in the cache. When good balance is not achieved, the hashing function would be ineffective and would cause conflict misses. To measure the balance we use a formula suggested by Aho, et al. [1]:

$$balance = \frac{\sum_{j=1}^{n_{set}} \frac{b_j \cdot (b_j+1)}{2}}{\frac{m}{2 \times n_{set}} \cdot (m + 2 \times n_{set} - 1)} \qquad (1)$$

where b_j represents the total number of addresses that are mapped to set j. $\frac{b_j \cdot (b_j+1)}{2}$ represents the weight of the set j, equivalent to $1 + 2 + ... + b_j$. Thus, a set that has more addresses will have a larger weight. The numerator, $\sum_{j=1}^{n_{set}} \frac{b_j \cdot (b_j+1)}{2}$, represents the sum of the weights of all sets. The denominator, $\frac{m}{2 \times n_{set}} \cdot (m + 2 \times n_{set} - 1)$, represents the sum of the weights of all sets, but assuming a perfectly random address distribution across all sets. Thus, a lower $balance$ value, with an ideal value of 1, represents better address distribution across the sets.

Concentration is a less straightforward measure and is intended to measure how evenly the sets are used over small intervals of accesses. It is possible to achieve the ideal balance for the entire address sequence $< a_1, a_2, ..., a_m >$, yet conflicts can occur if on smaller intervals the balance is not achieved.

To measure concentration, we calculate the distance d_i as the number of accesses to the cache that occur between two accesses to a particular set and calculate the standard deviation of these distances. More formally, d_i for an address a_i is the smallest positive integer such that

$H(a_i) = H(a_{i+d_i})$. The concentration is equal to the standard deviation of d_i's. Noting that in the ideal case, and with the balance of 1, the average of d_i's is necessarily n_{set}[1], our formula is

$$concentration = \sqrt{\frac{\sum_{i=1}^{m}(d_i - n_{set})^2}{m}} \qquad (2)$$

Using standard deviation penalizes a hashing function not only for re-accessing a set after a small time period since its last access ($d_i < n_{set}$), but also for a large time period ($d_i > n_{set}$). Similar to the balance, the smaller the concentration is, the better the hashing function is. The concentration of an ideal hashing function is zero.

In general, alternative hashing functions have mostly targeted the ideal balance, but not the ideal concentration. Achieving good concentration is vital to avoid the pathological behavior for applications with high temporal locality. Assume that a set receives a burst of accesses that consist of distinct addresses, then, the set suffers from conflict misses temporarily. If one of the addresses has temporal reuse, it may have been replaced from the cache by the time it is re-accessed, creating conflict misses.

2.2. Ideal Properties

In this section, we describe the properties that should be satisfied by an ideal hashing function. Most applications, even some irregular applications, often have strided access patterns. Given the common occurrence of these patterns, a hashing function that does not achieve the ideal balance and concentration will cause a *pathological behavior*. A pathological behavior arises when the balance or concentration of an alternative hashing function is worse than those of the traditional hashing function, often leading to slowdown.

Let s be a stride amount in the address sequence $< a_1, a_2, ..., a_m >$, i.e., $a_{i+1} = a_i + s$ where $1 \leq i < m$.

Property 1: Ideal balance. For the modulo based hashing where $H(a_i) = a_i$ **mod** n_{set}, the ideal balance is achieved if and only if $gcd(s, n_{set}) = 1$. For other hashing functions, such as XOR-based, the ideal balance condition is harder to formulate because the hashing function has various cases where the ideal balance is not achieved (Section 3.3).

Property 2: Sequence invariance. A hashing function is *sequence invariant* if and only if for any a_i, $H(a_i) = H(a_{i+x})$ implies $H(a_{i+1}) = H(a_{i+x+1})$.

The ideal concentration is achieved when *both* the ideal balance and sequence invariance are satisfied. Therefore, the ideal concentration is not achieved when the sequence invariance is not achieved. The sequence invariance says that once a set is re-accessed, the sequence of set accesses will precisely follow the previous sequence. Moreover, when the sequence invariance is satisfied, all the distances between two accesses to the same set are equal to a constant d, indicating the absence of a burst of accesses to a single set for the strided access pattern. Furthermore, when the ideal balance is satisfied for the modulo hashing, then the constant d is the average distance, or $d = x = n_{set}$.

It is possible that a hashing function satisfies the sequence invariance property in most, but not all, cases. Such a function can be said to have *partial sequence invariance*.

In Section 3.3, we will show that prime modulo hashing function satisfies both properties except for a very small number of cases, whereas other hashing functions do not always achieve Property 1 and 2 simultaneously. Bad concentration is a major source of the pathological behavior for alternative hashing functions.

3. Hashing Functions Based on Prime Numbers

In this section, we describe the prime modulo and prime displacement hashing functions that we propose (Section 3.1 and 3.2, respectively). We compare them against other hashing functions in Section 3.3

3.1. Prime Modulo Hashing Function

Prime modulo hashing functions, like any other modulo functions, can be expressed as $H(a_i) = a_i \bmod n_{set}$. The difference between prime modulo hashing and traditional hashing functions is that n_{set} is a prime number instead of a power of two. In general, n_{set} is the largest prime number that is smaller than a power of two. The prime modulo hashing functions that have been used in software hash tables [1] and the BSP machine [10], have two major drawbacks. First, they are considered expensive to implement in hardware because performing a modulo operation with a prime number requires an integer division [10]. Second, since the number of sets in the physical memory (n_{set_phys}) is likely a power of two, there are $\Delta = n_{set\_phys} - n_{set}$ sets that are wasted, causing fragmentation. For example, the fragmentation in BSP is a non-trivial 6.3%.

Since we target the L2 cache, however, this fragmentation becomes negligible. Table 1 shows that the percentage of the sets that are wasted in an L2 cache is small for commonly used numbers of the sets in the L2 cache. The fragmentation falls below 1% when there are 512 physical sets or more. This is due to the fact that there is always a prime number that is very close to a power of two.

[1] There are m accesses spread ideally across n_{set} sets, so the total distance between accesses is $m \cdot n_{set}$. Hence, the average over the m accesses is n_{set}

n_{set_phys}	n_{set}	Fragmentation (%)
256	251	1.95%
512	509	0.59%
1024	1021	0.29%
2048	2039	0.44%
4096	4093	0.07%
8192	8191	0.01%
16384	16381	0.02%

Table 1: Prime modulo set fragmentation.

Utilizing number theory we can compute the prime modulo function quickly without using an integer division. The foundation for the technique is taken from fast random number generators [13, 24]. Specifically, computing $2k \cdot x$ **mod** m where m is a Mersenne prime number (i.e., m is one less than a power of two) can be performed using add operations without a multiplication or division operation.

Since we are interested in a prime number n_{set} that is not necessarily Mersenne prime, we extend the existing method and propose two methods of performing the prime modulo operation fast without any multiplication and division. The first method, *iterative linear*, needs recursive steps of shift, add, and subtract&select operations. The second method, *polynomial*, needs only one step of add, and a subtract&select operation.

Subtract&select method. Computing the value of x **mod** n_{set} is trivial if x is small. Figure 2 shows how this can be implemented in hardware. x, $x - n_{set}$, $x - 2n_{set}$, $x - 3n_{set}$, etc. are all fed as input into a selector, which chooses the rightmost input that is not negative. To implement this method, the maximum value of x should be known.

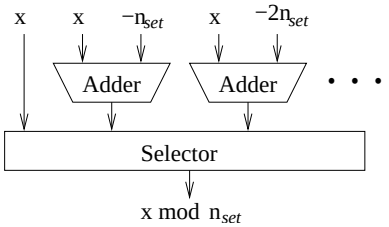


Figure 2: Hardware implementation of the subtract&select method.

Iterative linear method. First, a_i is represented as a linear function of $\Delta = n_{set\_phys} - n_{set}$. To see how this can be done, let T_i and x_i represent parts of the bits in a_i as depicted in Figure 1. Since $a_i = n_{set\_phys} \cdot T_i + x_i$, then

$$\begin{aligned} a_i &\equiv n_{set\_phys} \cdot T_i + x_i \ (\textbf{mod}\ n_{set}) \\ &\equiv (n_{set} + \Delta) \cdot T_i + x_i \ (\textbf{mod}\ n_{set}) \\ &\equiv \Delta \cdot T_i + x_i \ (\textbf{mod}\ n_{set}) \\ &\equiv a_i' \ (\textbf{mod}\ n_{set}) \end{aligned} \quad (3)$$

Since a_i' is much smaller than a_i, a_i' **mod** n_{set} may be computed using the subtract&select method (Figure 2). Moreover, although equation 3 contains a multiplication, since Δ is a very small integer (at most 9, see Table 1) for most cases, the multiplication can easily be converted to shift and add operations. For example, when $\Delta = 9$, $a_i \equiv T_i << 3 + T_i + x_i \ (\textbf{mod}\ n_{set})$, where $<<$ denotes a left shift operation. Finally, when a_i' is still large, we can apply Equation 3 iteratively to obtain a_i'', a_i''', etc. The following theorem states the maximum number of iterations that needs to be performed to compute a cache index using the iterative linear method.

Theorem 1. Given a B-bit address and a cache with block/line size of L, the number of iterations needed to compute a_i **mod** n_{set} is,

$$\left\lceil \frac{B - \log_2 L - \log_2 n_{set}}{\log_2 n_{set\_phys} - \log_2 \Delta} \right\rceil$$

When a subtract&select with $2^t + 2$ selector inputs is used in conjunction with the iterative linear method, the number of iterations required becomes,

$$\left\lceil \frac{B - \log_2 L - \log_2 n_{set}}{t + \log_2 n_{set\_phys} - \log_2 \Delta} \right\rceil$$

For example, for a 32-bit machine with $n_{set\_phys} = 2048$ and a 64-byte cache line size, the prime modulo can be computed with only two iterations. However, with a 64-bit machine, it requires 6 iterations using a subtract&select with 3-input selector, but requires 3 iterations with a 258-input selector.

Polynomial method. Because in some cases the iterative linear method involves multiple iterations, we devise an algorithm to compute the prime modulo operation in one step. To achieve that, using the same method as in deriving Equation 3, we first express a_i as a polynomial function of n_{set_phys}:

$$\begin{aligned} a_i \equiv\ & x_i + t_{i1} \cdot n_{set\_phys} + t_{i2} \cdot n_{set\_phys}^2 + \ldots \\ & + t_{in} \cdot n_{set\_phys}^n \quad (\textbf{mod}\ n_{set}) \end{aligned}$$

where t_{ij} consists of bit $log_2 n_{set\_phys} \cdot j$ through bit $log_2 n_{set\_phys} \cdot (j+1) - 1$ of the address bits of a_i. For example, t_{i1} is shown as t_i in Figure 1. Substituting n_{set_phys} by $(n_{set} + \Delta)$, we obtain

$$\begin{aligned} a_i \equiv\ & x_i + t_{i1} \cdot (n_{set} + \Delta) + t_{i2} \cdot (n_{set} + \Delta)^2 + \ldots \\ & + t_{in} \cdot (n_{set} + \Delta)^n \quad (\textbf{mod}\ n_{set}) \end{aligned}$$

$(n_{set} + \Delta)^k$, where $k = 1, 2, \ldots, n$, can be expanded into

$$\frac{k!}{k!0!} n_{set}^k + \frac{k!}{(k-1)!1!} n_{set}^{(k-1)} \Delta^1 + \ldots + \Delta^k$$

In the $(\textbf{mod}\ n_{set})$ space, any term that is a multiple of n_{set} is equivalent to zero. Since only the last term is not

zero, then $(n_{set} + \Delta)^k \equiv \Delta^k \ (\mathbf{mod}\ n_{set})$. Therefore, we can express a_i as a polynomial function of Δ:

$$\begin{aligned} a_i &\equiv x_i + t_{i1} \cdot \Delta + t_{i2} \cdot \Delta^2 + \ldots + t_{in} \cdot \Delta^n \\ &\quad (\mathbf{mod}\ n_{set}) \\ &\equiv a_i^* \quad (\mathbf{mod}\ n_{set}) \end{aligned} \tag{4}$$

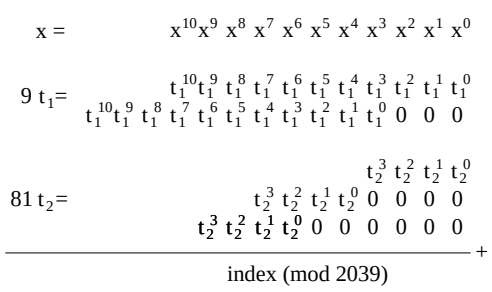
Note that a_i^* is much smaller than a_i, and is in general small enough to derive the result of the prime modulo using the subtract&select method (Figure 2).

A special but restrictive case is when n_{set} is a Mersenne prime number, in which case $\Delta = 1$. Then, Equation 4 can be simplified further, leading up to the same implementation as in [25]:

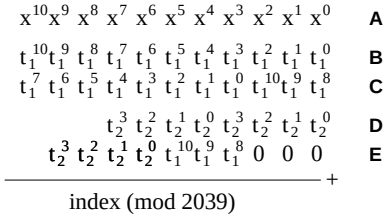
$$a_i \equiv x_i + t_{i1} + t_{i2} + \ldots + t_{in} \quad (\mathbf{mod}\ n_{set}) \tag{5}$$

It is possible to use n_{set} that is equal to $n_{set\_phys} - 1$ but not a prime number. Often, if $n_{set\_phys} - 1$ is not a prime number, it is a product of two prime numbers. Thus, it is at least a good choice for most stride access patterns. However, it is beyond the scope of this paper to evaluate such numbers.

Comparing the iterative linear and polynomial methods, the polynomial method allows smaller latency in computing the prime modulo when Δ is small, especially for 64-bit machines and a small number of sets in the cache. The iterative linear method is more desirable for low hardware and power budget, or when Δ is large.



(a)



(b)

Figure 3: The initial components of index calculation (a), and the components after optimizations (b).

3.1.1. Hardware Implementation

To illustrate how the prime modulo indexing can be implemented in hardware, let us consider an L2 cache with 64 byte blocks and 2048 ($= 2^{11}$) number of physical sets and 2039 ($= 2^{11} - 9$) number of sets. Therefore, 6 bits are used as the block offset, while the tag can be broken up into three components: 11-bit x ($x^{10}, x^9, x^8, \ldots, x^0$), 11-bit t_1 ($t_1^{10}, t_1^9, t_1^8, \ldots, t_1^0$), and the remaining 4-bit t_2 (t_2^3, t_2^2,t_2^1, t_2^0). According to Equation 4, the cache index can be calculated as $index = x + 9 \cdot t_1 + 81 \cdot t_2$. The binary representation of 9 and 81 are '1001' and '1010001', respectively. Therefore, Figure 3a shows that the index can be calculated as the sum of six numbers. This can be simplified further. The highest three bits of the third number ($t_1^{10} \cdot 2^{13} + t_1^9 \cdot 2^{12} + t_1^8 \cdot 2^{11}$) can be separated and according to Equation 4, is equal to $9 \cdot (t_1^{10} \cdot 2^2 + t_1^9 \cdot 2^1 + t_1^8 \cdot 2^0)$ in the modulo space. Furthermore, some of the numbers can be added instantly to fill in the bits that have zero values. For example, the fourth and the fifth numbers are combined into a single number. The resulting numbers are shown in Figure 3b. There are only five numbers (**A** through **E**) that need to be added, with up to 11 bits each.

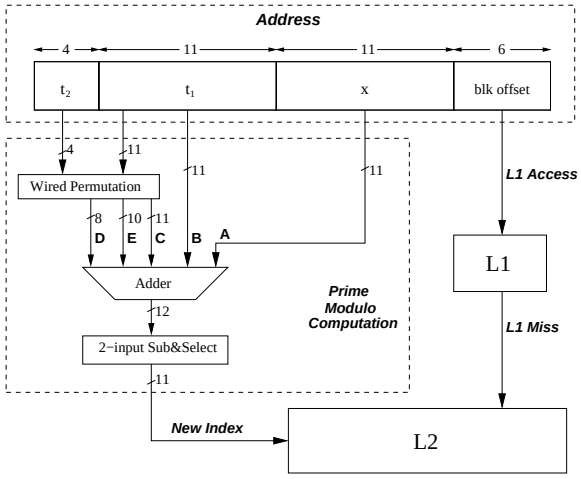


Figure 4: Hardware implementation of the prime modulo hashing using the polynomial method for $n_{set} = 2039$ on a 32-bit machine

Figure 4 shows how the prime modulo hashing computation fits with the overall system. The figure shows how the *new index* is computed from the addition of the five numbers (**A** through **E** in Figure 3b). **A** and **B** are directly taken from the address bits, while **C**, **D**, and **E** are obtained by wired permutation of the tag part of the address. The sum of the five numbers is then fed into a subtract&select with a 2-input selector. Only 2 inputs are needed by the selector because the maximum value of the addition can only be slightly larger than 2039. The reason for this is that in computing the addition, any intermediate carry out can be converted to 9 and be added to other numbers (Equation 4).

The figure also shows that the prime modulo computation can be overlapped with L1 cache accesses. On each L1 cache access, the prime modulo L2 cache index is computed. If the access results in an L1 cache miss, the new L2 cache index has been computed and is ready for use. The prime modulo computation is simple enough that the L2 access time is not likely impacted.

If a TLB is accessed in parallel with the L1 cache, the prime modulo computation can be partially cached in the TLB, simplifying the computation further. A physical address consists of the page index and the page offset. To compute the prime modulo L2 cache index, we can compute the page index modulo independently from the page offset modulo. On a TLB miss, the prime modulo of the missed page index is computed and stored in the new TLB entry. This computation is not in the critical path of TLB access, and does not require modifications to the OS' page table. On an L1 miss, the pre-computed modulo of the page index is added with the page offset bits that are not part of L2 block offset. For example, if the page size is 4KB, then only $12 - 6 = 6$ bits of the page offset needs to be added to the 11-bit pre-computed modulo, followed by a subtract&select operation to obtain the L2 cache index. This is a very simple operation that can probably be performed in much less than one clock cycle.

3.2. Prime Displacement Hashing Function

In the prime displacement hashing, the traditional modulo is performed after an offset is added to the original index x_i. The offset is the product of a number p and the tag T_i.

$$H(a_i) = (p \cdot T_i + x_i) \textbf{ mod } n_{set} \quad (6)$$

This new hashing function is based on hashing functions in Aho, et al. [1], and is related to Raghavan's RANDOM-H functions [14], with the main difference being their use of non-constant offset, resulting in not satisfying the sequence invariance property. Prime displacement hashing functions can easily be implemented in hardware with a narrow truncated addition if a prime number with few 1's in its binary representation is used as p [2].

One advantage of the prime displacement hashing function compared to the prime modulo hashing function is that the complexity of calculating the cache index in the prime displacement hashing function is mostly independent of the machine sizes. This makes it trivial to implement in machines with 64-bit or larger addressing.

[2] We had originally considered that a prime number would work best for p, hence the name *Prime Displacement* was introduced. But, technically an odd number modulo a power of two forms a modulo multiplication group M_m [17] and as such they are all invertible so none of them are prime in a technical sense. In practice, it is also not the case that prime numbers are necessarily better choices for *p* than ordinary odd numbers.

3.3. Comparison of Various Hashing Functions

Before we compare our hashing functions with existing ones, we briefly overview existing hashing functions. The *traditional hashing* function H_{trad} is a very simple modulo based hashing function. It can be expressed as $H_{trad} = x_i$ or equivalently as $H_{trad}(a_i) = a_i$ **mod** n_{set_phys}. Pseudo-random hashing randomizes accesses to cache sets. Examples of the pseudo-random hashing are *XOR*-based hashing functions, which are by far the most extensively studied [15, 16, 23, 22, 26, 11, 18, 4, 19]. We choose one of the most prominent examples: $H(a_i) = t_i \bigoplus x_i$, where $\bigoplus$ represents the bitwise exclusive OR operator. In a *skewed associative cache*, the cache itself is divided into banks and each bank is accessed using a different hashing function. Here, cache blocks that are mapped to the same set in one bank are most likely not to map to the same set in the other banks. Seznec proposes using an XOR hashing function in each bank after a circular shift is performed to the bits in t_i [18, 4, 19]. The number of circular shifts performed differs between banks. This results in a form of a perfect shuffle. We propose using the prime displacement hashing functions in a skewed cache. To ensure inter-bank dispersion, a different prime number for each bank is used.

Table 2 compares the various hashing functions based on the following:

- when the ideal balance is achieved,
- whether they satisfy the *sequence invariance* property,
- whether a simple hardware implementation exists, and
- whether they place restrictions on the replacement algorithm.

The major disadvantage of the traditional hashing is that it achieves the ideal balance only when the stride amount s is odd, where $gcd(s, n_{set}) = 1$. When the ideal balance is satisfied, however, it achieves the ideal concentration because it satisfies the sequence invariance property. Note that for a common unit stride ± 1, it has the ideal balance and concentration. Thus, any hashing functions that achieve less than the ideal balance or concentration with unit strides are bound to have a pathological behavior.

XOR achieves the ideal balance on most stride amounts s, but always has less than the ideal concentration because it does not satisfy the sequence invariance property. This is because the sequence of set accesses is never repeated due to XOR-ing with different values in each subsequence.

Characteristics	Single Hashing Function				Multiple Hashing Functions	
	Traditional	XOR	pMod	pDisp	Skewed	Skewed + pDisp
Ideal balance condition	s odd	various	all s except $s = k \times n_{set}$	Most odd, all even s	None	None
Sequence invariant?	Yes	No	Yes	Partial	No	No
Simple Hw Impl.	Yes	Yes	Yes	Yes	Yes	Yes
Replacement Alg. Restriction	No	No	No	No	Yes	Yes

Table 2: Qualitative comparison of various hashing functions: traditional, XOR, prime modulo (pMod), prime displacement (pDisp), skewed associative cache with circular-shift and XOR (Skewed) [18, 4, 19], and our skewed associative cache with prime displacement (Skewed + pDisp).

Thus, it is prone to the pathological behavior. There are various cases where the XOR hashing does not achieve the ideal balance. One case is when $s = n_{set}-1$. For example, with $s = 15$ and $n_{set} = 16$ (as in a 4-way 4KB cache with 64 byte lines), it will access sets 0, 15, 15, 15, ... Not only that, a stride of 3 or 5 will also fail to achieve the ideal balance because they are factors of 15. This makes the XOR a particularly bad choice for indexing the L1 cache.

The prime modulo hashing (*pMod*) achieves the ideal balance and concentration except for very few cases. The ideal balance is achieved because $gcd(s, n_{set}) = 1$ except when s is a multiple of n_{set}. The ideal concentration is achieved because $H(a_i) = H(a_{i+d})$ implies that $H(a_{i+1}) = H(a_i + s) = H(a_{i+d} + s) = H(a_{i+d+1})$ with the stride amount s. This makes the prime modulo hashing an ideal hashing function. As we have shown in Section 3.1, fast hardware implementations exist.

The prime displacement hashing (*pDisp*) achieves an ideal balance with even strides and most odd strides. Although it does not satisfy the sequence invariance property, the distance between two accesses to the same set is almost always constant. That is, for all but one set in a single subsequence, $H(a_i) = H(a_{i+x})$ implies $H(a_{i+1}) = H(a_{i+x+1})$. Furthermore, $x = n_{set} - p$, where p is the prime number used. Thus, it partially satisfies the sequence invariance.

Skewed associative caches do not guarantee the ideal balance or concentration, whether they use the XOR-based hashing (*Skewed*), or the prime displacement hashing (*Skewed+pDisp*). However, probabilistically, the accesses will be quite balanced since an address can be mapped to a different place in each bank. A disadvantage of skewed associative caches is the fact that they make it difficult to implement a least recently used (LRU) replacement policy and force using pseudo-LRU policies. The non-ideal balance and concentration, together with the use of a pseudo-LRU replacement policy, make a skewed associative cache prone to the pathological behavior, although it works well on average. Later, we will show that skewed caches do degrade the performance of some applications.

Finally, although not described in Table 2, some other hashing functions, such as all XOR-based functions and random-h [14, 15, 16, 20, 8, 7, 26, 11], are not sequence invariant, and therefore do not achieve the ideal concentration.

4. Evaluation Environment

Applications. To evaluate the prime hashing functions, we use 23 memory-intensive applications from various sources: bzip2, gap, mcf, and parser from Specint2000 [21], applu, mgrid, swim, equake, and tomcatv from Specfp2000 and Specfp95 [21], mst from Olden, bt, ft, lu, is, sp, and cg from NAS [12], sparse from Sparsebench [6], and tree from the University of Hawaii [3]. Irr is an iterative PDE solver used in CFD applications. Charmm is a well known molecular dynamics code and moldyn is its kernel. Nbf is a kernel from the GROMOS molecular dynamics benchmarks. And, euler is a 3D Euler equation solver from NASA.

We categorize the applications into two groups: a group where the histogram of number of accesses to different sets are uniform, and those that are not uniform. Let $f_1, f_2, \ldots, f_{n_{set}}$ represent the frequency of accesses to the sets $1, 2, \ldots, n_{set}$ in the L2 cache. An application is considered to have a non-uniform cache access behavior if the ratio $stdev(f_i)/\bar{f}$ is greater than 0.5. Applications with non-uniform cache set accesses likely suffer from conflict misses, and hence alternative hashing functions are expected to speed them up.

Among the 23 applications, we found that 30% of them (7 benchmarks) are non-uniform: bt, cg, ft, irr, mcf, sp, and tree.

Simulation Environment. The evaluation is done using an execution-driven simulation environment that supports a dynamic superscalar processor model [9]. Table 3 shows the parameters used for each component of the architecture. The architecture is modeled cycle by cycle.

Prime Numbers. The prime modulo function uses the prime number shown in Table 1. The prime displacement function uses a number 9 when it is used as a single hashing function. When used in conjunction with a skewed associative cache the numbers that are used for each of the four cache banks are 9, 19, 31, and 37. Although the number 9 is not prime, it is selected for the reason explained in

PROCESSOR
6-issue dynamic. 1.6 GHz. Int, fp, ld/st FUs: 4, 4, 2 Pending ld, st: 8, 16. Branch penalty: 12 cycles
MEMORY
L1 data: write-back, 16 KB, 2 way, 32-B line, 3-cycle hit RT L2 data: write-back, 512 KB, 4 way, 64-B line, 16-cycle hit RT RT memory latency: 243 cycles (row miss), 208 cycles (row hit) Memory bus: split-transaction, 8 B, 400 MHz, 3.2 GB/sec peak Dual channel DRAM. Each channel: 2 B, 800 MHz. Total: 3.2 GB/sec peak Random access time (*tRAC*): 45 ns Time from memory controller (*tSystem*): 60 ns

Table 3: Parameters of the simulated architecture. Latencies correspond to contention-free conditions. *RT* stands for round-trip *from the processor*.

Section 3.2.

5. Evaluation

In this section, we present and discuss six sets of evaluation results. Section 5.1 shows the balance and concentration for four different hashing functions. Section 5.2 presents the results of using a single hashing function for the L2 cache, while Section 5.3 presents the results of using multiple hashing functions in conjunction with the skewed associative L2 cache. Section 5.4 provides an overall comparison of the hashing functions. Section 5.5 shows the miss reduction achieved by the hashing functions. Finally, Section 5.6 presents cache miss distribution of tree before and after applying the prime modulo hashing function.

5.1. Balance and Concentration

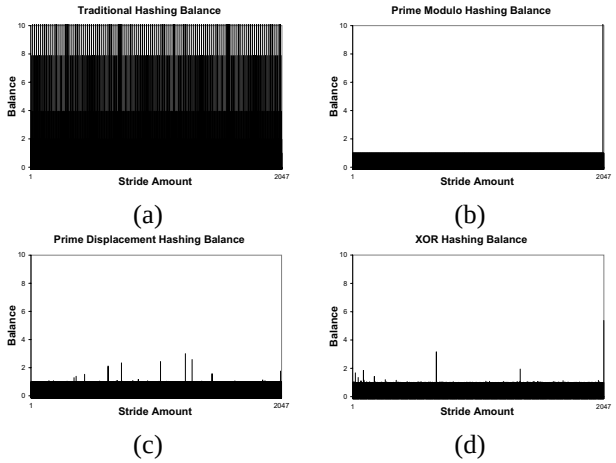


Figure 5: Balance for the Traditional Hashing (a), Prime Modulo Hashing (b), Prime Displacement Hashing (c), and XOR Hashing (d)

Figure 5 shows the balance values of the four different hashing functions using a synthetic benchmark that produces only strided access patterns. The stride size is varied from 1 to 2047. The maximum balance displayed in the vertical axes in the figure is limited to 10 for easy comparison. Note that small strides are more likely to appear in practice. Therefore, they are more important than large strides. The balance values for the traditional and prime modulo hashing functions follow the discussion in Section 3.3. In particular, the traditional hashing function suffers from bad balance values with even strides, but achieves perfect balance with odd strides. The prime modulo achieves perfect balance, except when the stride is equal to n_{set}, which in this case is 2039. XOR and Prime displacement hashing also have the ideal balance with most strides. Both have various cases in which the stride size causes non-ideal balance. The non-ideal balance is clustered around the small strides with the XOR function, whereas it is concentrated toward the middle for the prime displacement function. Thus, in practice, the prime displacement is superior to the XOR hashing function.

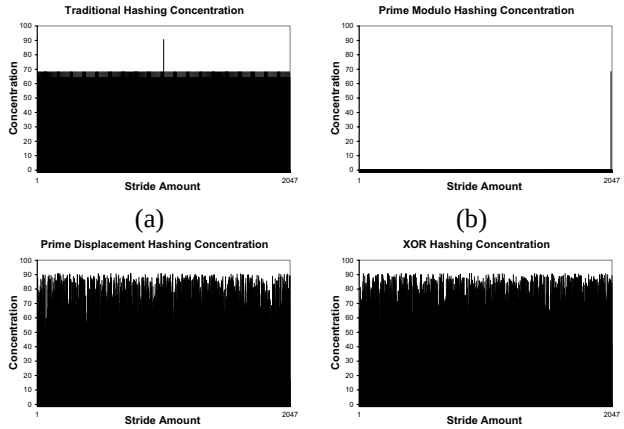


Figure 6: Concentration: Traditional Hashing (a), Prime Modulo Hashing (b), Prime Displacement Hashing (c), and XOR Hashing (d)

Figure 6 shows the concentration for the same range of stride size. As expected, the traditional hashing function suffers from very bad concentration with even strides, but achieves perfect concentration with odd strides. The XOR and prime displacement hashing functions also suffer from bad concentration for many strides. This is because prime displacement hashing is only partially sequence invariant. Since the prime modulo hashing is sequence invariant, it achieves ideal concentration except for the stride equal to n_{set}. Hence, for strided accesses, we can expect the prime modulo hashing to have the best performance between our four hashing functions. More importantly, the prime modulo hashing also achieves ideal concentration with odd strides the same way as the traditional hashing. Hence, we can expect that the prime modulo hashing is

resistant to the pathological behavior. The XOR hashing function may outperform the traditional hashing on average. However, it cannot match the ideal concentration of the traditional hashing with odd stride amount. Thus it is prone to the pathological behavior.

5.2. Single Hashing Function Schemes

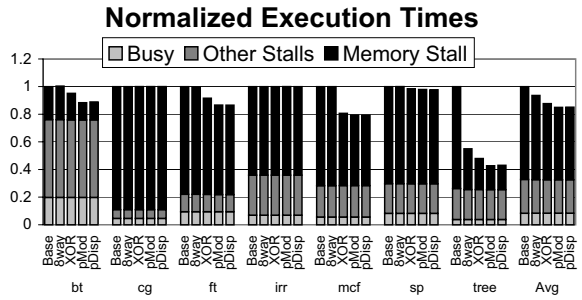


Figure 7: Performance results of single hashing functions for applications with non-uniform cache accesses

Figure 7 and 8 show the execution time of each application with non-uniform cache accesses and uniform cache accesses, respectively. Each figure compares the execution time with different hashing functions: a traditional hashing function with 4-way associative L2 cache (*Base*), a traditional hashing function with an 8-way associative same-size L2 cache (*8-way*), the XOR hashing function (*XOR*), the prime modulo hashing function (*pMod*), and the prime displacement hashing function (*pDisp*) as described in Section 3.3. The execution time of each case is normalized to *Base*. All bars are divided into: execution of instructions (*Busy*), stall time due to various pipeline hazards (*Other Stalls*), and stall time due to memory accesses (*Memory Stall*).

Both Figure 7 and 8 show that increasing the L2 cache associativity to 8-way only improves the execution time marginally, with the exception of tree. This is mainly because doubling associativity but keeping the same cache size reduces the number of sets in half. In turn, this doubles the number of addresses mapped to a set. Thus, increasing cache associativity without increasing the cache size is not an effective method to eliminate conflict misses.

Comparing the *XOR*, *pMod*, and *pDisp* for applications with non-uniform cache accesses in Figure 7, they all improve execution time significantly, with both *pMod* and *pDisp* achieving an average speedup of 1.27. It is clear that *pMod* and *pDisp* perform the best, followed by the *XOR*. Although *XOR* is certainly better than *Base*, its non-ideal balance for small strides and its non-ideal concentration hurt its performance such that it cannot obtain optimal speedups. For applications that have uniform cache accesses in Figure 8, the same observation generally holds. However, *8-way* slows down mst by 2%, and *XOR* and *pMod* slow down sparse by 2%. Some other applications have slight speedups.

5.3. Multiple Hashing Functions Schemes

Figure 9 and 10 show the execution times of applications with non-uniform cache accesses and uniform cache accesses, respectively when multiple hashing functions are used. Each figure compares the execution time with different hashing functions: a traditional hashing function with 4-way associative L2 cache (*Base*), the prime modulo hashing that is the best single hashing function from Section 5.2 (*pMod*), the XOR-based skewed associative cache proposed by Seznec [19] (*SKW*), and the skewed associative cache with the prime displacement function that we propose (*skw+pDisp*) as described in Section 3.3. The execution time of each case is normalized to *Base*.

The skewed associative caches (*SKW* and *skw+pDisp*) are based on Seznec's design that uses four direct-mapped cache banks. The replacement policy is called Enhanced Not Recently Used (ENRU) [19]. The only difference between *SKW* and *skw+pDisp* is the hashing function used (XOR versus prime displacement). We have also tried a different replacement policy called NRUNRW (Not Recently Used Not Recently Written) [18]. We found that it gives similar results.

Figure 9 and 10 show that *pMod* sometimes outperforms and is sometimes outperformed by *SKW* and *skw+pDisp*. With cg and mst, only the skewed associative schemes are able to obtain speedups. This indicates that there are some misses that the strided access patterns cannot account for, and this type of misses cannot be tackled by a single hashing function. On average, *skw+pDisp* performs the best, followed by *SKW*, and then closely followed by *pMod*. The extra performance for the applications with non-uniform accesses, however, comes at the expense of having the pathological behavior for the applications that have uniform accesses (Figure 10). For example, *SKW* slows down six applications by up to 9% (bzip2, charmm, is, parser, sparse, and irr), while *skw+pDisp* slows down three applications by up to 7% (bzip2, mgrid, and sparse).

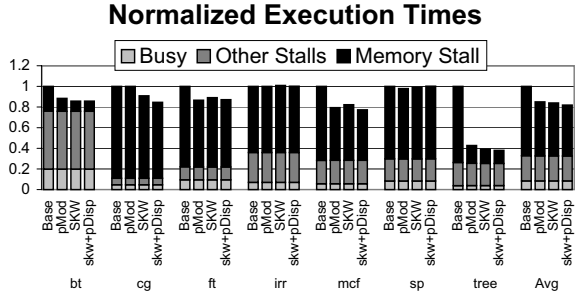


Figure 9: Performance results of multiple hashing functions for applications with nonuniform cache accesses

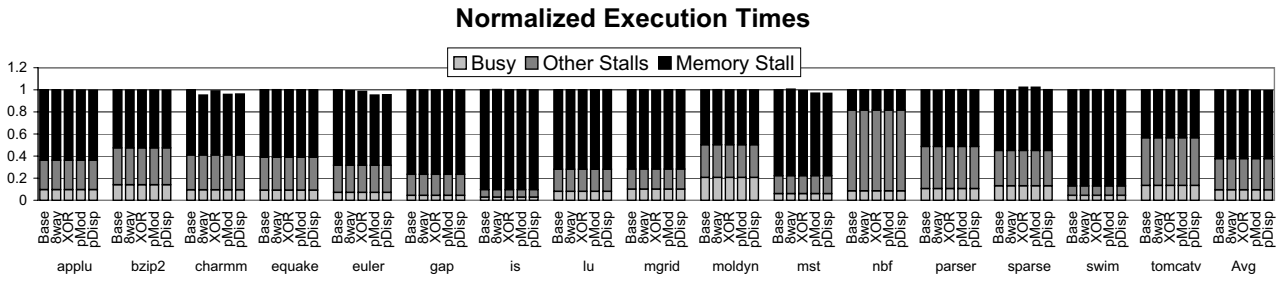


Figure 8: Performance results of single hashing functions for applications with uniform cache accesses

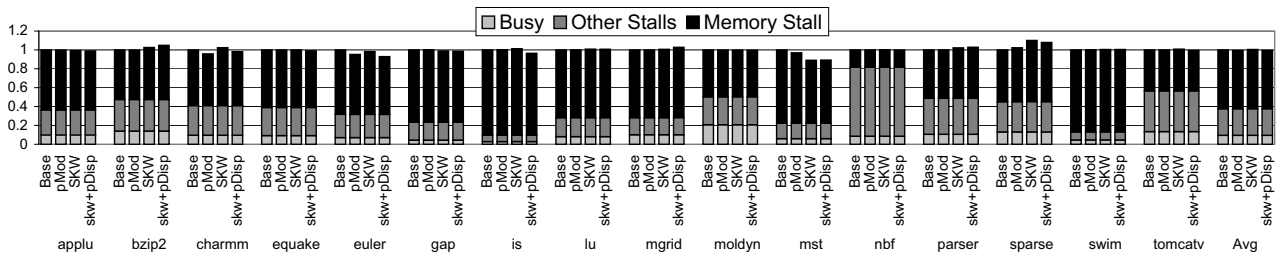


Figure 10: Performance results of multiple hashing functions for applications with uniform cache accesses

Cache Hashing	Uniform Apps min,avg,max	Nonuniform Apps min,avg,max	Patho. Cases
XOR	0.98,1.00,1.01	1.00,1.21.2.09	1
pMod	0.98,1.00,1.05	1.00,1.27,2.34	1
pDisp	1.00,1.01,1.05	1.00,1.27,2.32	0
SKW	0.91,1.00,1.12	0.99,1.31,2.55	4
skw+pDisp	0.93,1.01,1.12	1.00,1.35,2.63	4

Table 4: Summary of the Performance Improvement of the Different Cache Confi gurations

5.4. Overall Comparison

Table 4 summarizes the minimum, average, and maximum speedups obtained by all the hashing functions evaluated. It also shows the number of pathological behavior cases that result in a slowdown of more than 1% when compared to the traditional hashing function. In terms of average speedups, *skw+pDisp* is the best among the multiple hashing function schemes, while *pMod* is the best among the single hashing function schemes. In terms of pathological cases, however, the single hashing function schemes perform better than the multiple hashing function schemes. This is due to worse concentration and probably in part due to pseudo-LRU scheme used in skewed associative caches. In summary, the prime modulo and prime displacement hashing stand out as excellent hashing functions for L2 caches.

5.5. Miss Reduction

Figures 11 and 12 show the normalized number of L2 cache misses for the 23 benchmarks using a traditional hashing function (*Base*), the prime modulo hashing function (*pMod*), the prime displacement hashing function (*pDisp*), a skewed cache using prime displacement (*skw+pDisp*), and a fully-associative cache of the same size (*FA*). The number of misses are normalized to the (*Base*).

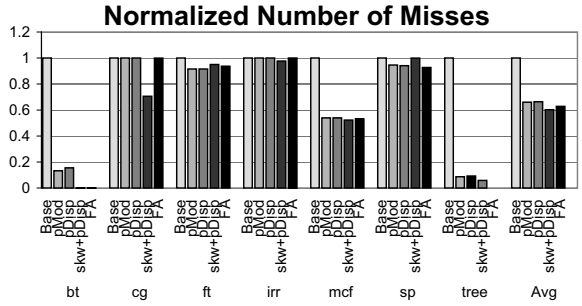


Figure 11: Normalized number of misses for several hashing functions for applications with non-uniform cache accesses

Figure 11 shows that our proposed hashing functions are on average able to remove more than 30% of the L2 cache misses. In some applications such as bt and tree, they eliminate nearly all the cache misses. Interestingly, *skw+pDisp* is able to remove more cache misses than a fully associative cache in cg, indicating that in some cases it can remove some capacity misses.

Figure 12 shows that most of the applications that have uniform cache accesses do not have conflict misses, and thus do not benefit from a fully-associative cache, except for charmm and euler. The figure highlights the *pMod*'s and *pDisp*'s resistance to pathological behavior. *pMod* does not increase the cache misses even for applications that already have uniform cache accesses due to its ideal balance and concentration. Although *pDisp* does not achieve an ideal concentration, its partial sequence invari-

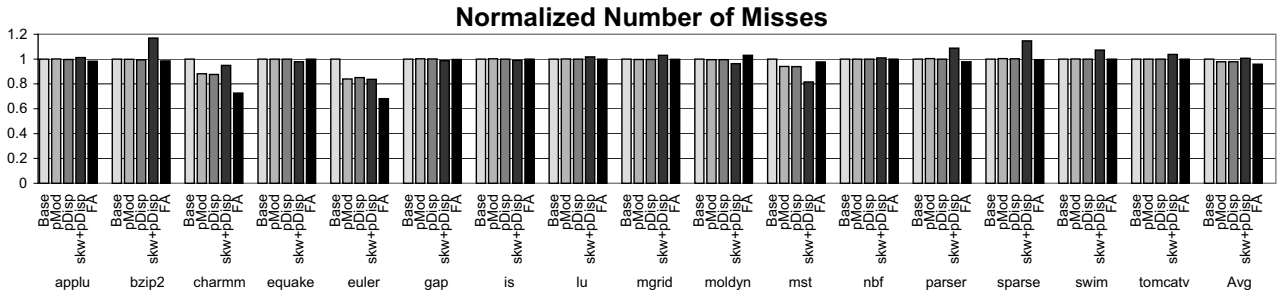


Figure 12: Normalized number of misses for several hashing functions for applications with uniform cache accesses

ance property helps it to achieve a very good concentration and in practice performs just as well as *pMod*. This is in contrast to *skw+pDisp*, which increases the number of misses by up to 20% in 6 applications (bzip2, mgrid, parser, sparse, swim, tomcatv). This again shows that skewed associative caches are prone to pathological behavior, although they are able to remove some capacity misses.

5.6. Cache Misses Distribution for tree

From previous figures, tree is the application where *pMod* and *pDisp* perform the best, both in terms of miss reduction and speedup. Figure 13 shows the distribution of L2 cache misses over the cache sets for tree using both traditional hashing and prime modulo hashing. In the traditional hashing case, Figure 13a shows that the vast majority of cache misses in tree are concentrated in about 10% of the sets. This is due to an unbalanced distribution of cache accesses in tree, causing some cache sets to be over-utilized and suffer from many conflict misses. By distributing the accesses more uniformly across the sets, *pMod* is able to eliminate most of those misses (Figure 13b).

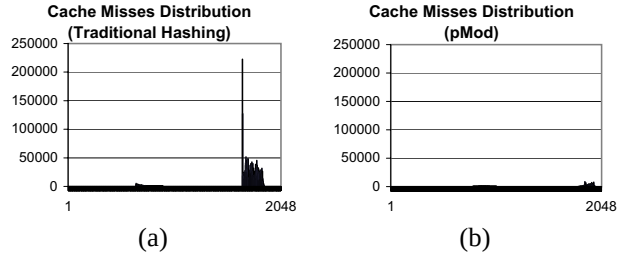


Figure 13: Distribution of misses across the cache sets for *tree* using (*Base*) hashing (a) and *pMod* hashing (b).

6. Related Work

Prior studies showed that alternative cache indexing/hashing functions are effective in reducing conflicts by achieving a more uniform access distribution across the L1 cache sets [18, 4, 22, 23], L2 cache sets [19], or the main memory banks [14, 10, 15, 16, 20, 8, 7, 26, 11]. Most of the prior hashing functions permute the accesses using some form of XOR operations. We found that XOR operations typically do not achieve an ideal concentration that is critical to avoiding pathological behavior under strided access patterns.

Although prime-based hashing has been proposed for software hash tables as in [1], its use in the memory subsystem has been very limited due to its hardware complexity that involves true integer division operations and fragmentation problems. Budnick and Kuck first suggested using a prime number of memory banks in parallel computers [5], which was later developed into the Borroughs Scientific Processor [10]. Yang and Yang proposed using Mersenne prime modulo hashing for cache indexing for vector computation [25]. Since Mersenne prime numbers are sparse, e.g. $31(2^5-1), 127(2^7-1), 8191(2^{13}-1), 131071(2^{17}-1), \ldots$, using them significantly restricts the number of cache sets that can be implemented. We derive a more general solution that does not assume Mersenne prime numbers and show that prime modulo hashing can be implemented fast on *any* number of cache sets. In addition, we present a prime displacement hashing function that achieves comparable performance and robustness to the prime modulo hashing function.

Compiler optimizations have also targeted conflict misses by padding the data structures of a program. One example is the work by Bacon, et. al. [2], who tried to find the optimal padding amount to reduce conflict misses in caches and TLBs in a loop nest. They tried to spread cache misses uniformly across loop iterations based on profiling information. Since the conflict behavior is often input dependent and determined at run time, their approach has limited applicability.

7. Conclusions

Even though using alternative cache indexing/hashing functions is a popular technique to reduce conflict misses by achieving a more uniform cache access distribution across the sets in the cache, no prior study has really analyzed the pathological behavior of such hashing functions that often result in performance degradation.

We presented an in-depth analysis of the pathological behavior of hashing functions and proposed two new hashing functions for the L2 cache that are resistant to the pathological behavior and yet are able to eliminate the worst case conflict behavior. The prime modulo hashing uses a prime number of sets in the cache, while the prime displacement hashing adds an offset that is equal to a prime number multiplied by the tag bits to the index bits of an address to obtain a new cache index. These hashing techniques can be implemented in fast hardware that uses a set of narrow add operations in place of true integer division and multiplication. This implementation has negligible fragmentation for the L2 cache. We evaluated our techniques with 23 applications from various sources. For applications that have non-uniform cache accesses, both the prime modulo and prime displacement hashing achieve an average speedup of 1.27, practically without slowing down any of the 23 benchmarks.

Although lacking the theoretical superiority of the prime modulo, when the prime number is carefully selected, the prime displacement hashing performs just as well in practice. In addition, the prime displacement hashing can easily be used in conjunction with a skewed associative cache, which uses multiple hashing functions to further distribute the cache accesses across the sets in the cache. The prime displacement hashing outperforms XOR-based hashing used in prior skewed associative caches. In some cases, the skewed associative L2 cache with prime displacement hashing is able to eliminate more misses compared to using a single hashing function. It shows an average speedup of 1.35 for applications that have non-uniform cache accesses. However, it introduces some pathological behavior that slows down four applications by up to 7%. Therefore, an L2 cache with our prime modulo or prime displacement hashing functions is a promising alternative.

References

[1] A. V. Aho and J. D. Ullman. *Principles of Compiler Design*, chapter 7.6, pages 434–8. Addison-Wesley, 1997.

[2] David F. Bacon, Jyh-Herng Chow, Dz ching R. Ju, Kalyan Muthukumar, and Vivek Sarkar. A Compiler Framework for Restructuring Data Declarations to Enhance Cache and TLB Effectiveness. In *Proceedings of CACON'94*, pages 270–282, October 1994.

[3] J. E. Barnes. Treecode. Institute for Astronomy, University of Hawaii. 1994. *ftp://hubble.ifa.hawaii.edu/pub/barnes/treecode*.

[4] F. Bodin and A. Seznec. Skewed-associativity improves performance and enhances predictability. *IEEE Transactions on Computers*, 1997.

[5] P. Budnick and D. J. Kuck. Organization and use of parallel memories. *IEEE Transactions on COmputers*, pages pp. 435–441, Dec 1971.

[6] J. Dongarra, V. Eijkhout, and H. van der Vorst. SparseBench: A Sparse Iterative Benchmark. *http://www.netlib.org/benchmark/sparsebench*.

[7] J. M. Frailong, W.Jalby, and J.Lenfant. Xor-schemes: a flexible data organization in parallel memories. In *Proceedings the International Conference on Parallel Processing*, 1985.

[8] D. T. Harper III and J. R. Jump. Vector access performance in parallel memories using a skewed storage scheme. In *IEEE Transactions on Computers*, Dec. 1987.

[9] V. Krishnan and J. Torrellas. A Direct-Execution Framework for Fast and Accurate Simulation of Superscalar Processors. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 286–293, October 1998.

[10] D. H. Lawrie and C. R. Vora. The prime memory system for array access. *IEEE Transactions on COmputers*, 31(5), May 1982.

[11] W. F. Lin, S. K. Reinhardt, and D. C. Burger. Reducing dram latencies with a highly integrated memory hierarchy design. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2001.

[12] NAS Parallel Benchmark. http://www.nas.nasa.gov/pubs/techreports/ nasreports/nas-98-009/.

[13] W. H. Payne, J. R. Rabung, and T. P. Bogyo. Coding the lehmer pseudo-random number generator. *Communication of ACM*, 12(2), 1969.

[14] R. Raghavan and J. Hayes. On randomly interleaved memories. In *Supercomputing*, 1990.

[15] B. R. Rau. Pseudo-randomly interleaved memory. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA)*, 1991.

[16] B. R. Rau, M. Schlansker, and D. Yen. The cydra 5 stride-insensitive memory system. In *Proceedings of the International Conference on Parallel Processing*, 1989.

[17] H. Riesel. *Prime Number and Computer Methods for Factorization 2nd Ed.*, pages pp. 270–2. Birkhauser, 1994.

[18] A. Seznec. A case for two-way skewed associative caches. In *Proceedings of the 20th International Symposium of Computer Architecture*, 1993.

[19] A. Seznec. A new case for skewed-associativity. *IRISA Technical Report #1114*, 1997.

[20] G. S. Sohi. Logical data skewing schemes for interleaved memories in vector processors. In *University of Wisconsin-Madison Computer Science Technical Report #753*, 1988.

[21] Standard Performance Evaluation Corporation. http://www.spec.org.

[22] N. Topham, A. Gonzalez, and J. Gonzalez. The design and performance of a conflict-avoiding cache. In *International Symposium on Microarchitecture*, pages 71–80, Dec. 1997.

[23] N. Topham, A. Gonzalez, and J. Gonzalez. Eliminating cache conflict misses through xor-based placement functions. In *International Conference on Supercomputing*, 1997.

[24] P.-C. Wu. Multiplicative, congruential random number generators with multiplier $\pm 2^{k1} \pm 2^{k2}$ and modulus $2^p - 1$. *ACM Transactions on Mathematical Software*, 23(2):255–65, 1997.

[25] Q. Yang and L. W. Yang. A novel cache design for vector processing. In *Proceedings of the International Symposium on Computer Architecture*, pages pp. 362–371, May 1992.

[26] Z. Zhang, Z. Zhu, and X. Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proceedings of the International Symposium on Microarchitecture*, 2000.