Trade-offs for Skewed-Associative Caches

Hans Vandierendonck[a]*and Koen De Bosschere[a]†

[a]Dept. of Electronics and Information Systems, Ghent University
Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium.

The skewed-associative cache achieves low miss rates with limited associativity by using inter-bank dispersion, the ability to disperse blocks over many sets in one bank if they map to the same set in another bank. This paper formally defines the degree of inter-bank dispersion and argues that high inter-bank dispersion conflicts with common micro-architectural designs in which the skewed-associative cache is embedded. Various trade-offs for the skewed-associative cache are analyzed and a skewed-associative cache organization with reduced complexity and a near-optimal cache miss rate is proposed.

1. Introduction

Cache memories hide the rapidly increasing memory latency by storing the most recently accessed data on-chip. Most implementations of caches are organized as a direct mapped or set-associative structure. In such an organization, every block of data may be stored in only a few locations of the cache. These locations form one set of the cache, hence the name "set-associative". The benefit of this approach is a fast cache access time, but the down-side is that many cache misses occur when many blocks are accessed that all map to the same set. These misses are called *conflict misses* and result from the set-associative cache organization. These misses occur frequently in scientific workloads and can significantly deteriorate performance [1, 2]. It was shown that the skewed-associative cache can remove these misses and improve the performance predictability [3].

The skewed-associative cache is an organization that combines a fast cache access time with few conflict misses [3, 4, 5]. A 2-way skewed-associative cache has a miss rate comparable to a 4-way set-associative cache [4]. An n-way skewed-associative cache has n direct mapped banks. Each bank is indexed using a different hash function. When many blocks map to the same set in one bank, then it is very unlikely that all of them map to the same set in *all* banks. The ability to spread blocks that map to the same set in one bank over multiple sets in the other banks is called *inter-bank dispersion* [4].

Ideally, inter-bank dispersion should be maximum, i.e., it should be possible to spread blocks over all sets in another bank. This paper shows that programs typically do not require such *maximum* inter-bank dispersion, but that a moderate amount of inter-bank dispersion suffices to remove nearly all conflict misses. Furthermore, we give some reasons why a moderate amount of inter-bank dispersion is desirable, from a micro-architectural point of view.

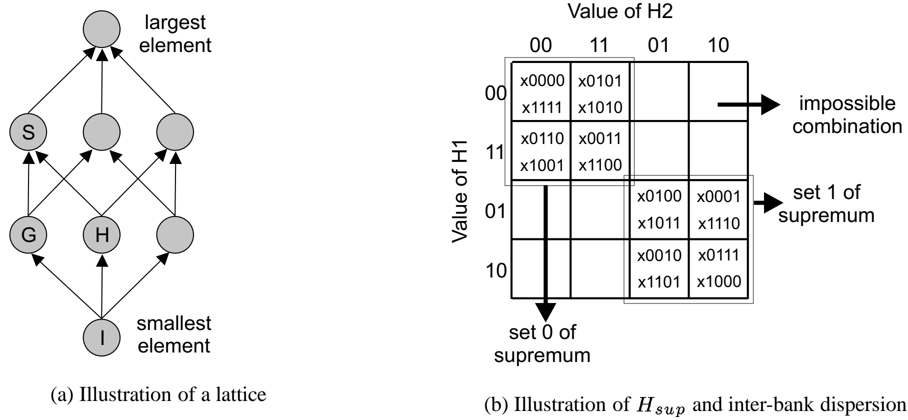
The remainder of this paper is organized as follows. In section 2, we present mathematical models of hash functions and formally define the *degree of inter-bank dispersion*. Then we describe in section 3 how various parameters of the skewed-associative cache affect the complexity of a skewed-associative cache organization. Section 4 presents a technique to construct conflict-avoiding hash functions with a pre-specified degree of inter-bank dispersion, number of hashed address bits and inputs per XOR. Using this technique, we evaluate trade-offs for the skewed-associative cache and propose a near-optimal configuration in section 5. Section 6 concludes this paper.

2. Mathematical Treatment

This section describes a mathematical model of a hash function and uses it to define the degree of inter-bank dispersion. These definitions are treated in detail in [6].

---

*Hans Vandierendonck is sponsored by the Flemish Institute for the Promotion of Scientifi c-Technological Research in the Industry (IWT).
He can be reached at hvdieren@elis.UGent.be
†Koen De Bosschere can be reached at kdb@elis.UGent.be



(a) Illustration of a lattice

(b) Illustration of H_{sup} and inter-bank dispersion

Figure 1. Illustration of the lattice of hash functions and an example.

2.1. Hash Functions

We represent an n-bit block address $\mathbf{a}$ by a bit vector $[a_{n-1}a_{n-2}\dots a_0]$, with a_{n-1} the most significant bit and a_0 the least significant bit. A hash function mapping n to m bits is represented as a binary matrix H with n rows and m columns. The bit on row r and column c is 1 when address bit a_r is an input to the XOR computing the c-th set index bit. Consequently, the computation of the set index $\mathbf{s}$ can be expressed as the vector-matrix multiplication over $GF(2)$, denoted by $\mathbf{s} = \mathbf{a}\,H$. $GF(2)$ is the domain $\{0, 1\}$ where addition is computed as XOR and multiplication is computed as logical *and*.

Every function in the design space of XOR-based set index functions can be represented by the *null space* of its matrix [7]. The null space $N(H)$ of a matrix H is the set of all vectors that are mapped to the zero vector:

$$N(H) = \{\mathbf{x} \in \{0,1\}^{1\times n} \mid \mathbf{x}\,H = \mathbf{0}\}.$$

The null space is a vector space with dimensionality $dim\ N(H) = n - m$. Conflict misses occur when $\mathbf{x}\,H = \mathbf{y}\,H$ or, $(\mathbf{x} \oplus \mathbf{y}) \in N(H)$, by noting that the XOR is its own inverse.

2.2. Set Refinement and the Lattice of Hash Functions

Set refinement is a relation between two set-associative caches. The relation holds when all addresses that map to the same set in one cache also map to the same set in the other cache [8]. This is expressed for XOR-based hash functions using null spaces as follows:

Definition 1 *For matrices H and G, H refines G ($H \sqsubseteq G$) iff $N(H) \subseteq N(G)$.*

E.g., if H maps two addresses $\mathbf{x}$ and $\mathbf{y}$ to the same set, then $(\mathbf{x} \oplus \mathbf{y}) \in N(H)$. If set refinement holds, then $(\mathbf{x} \oplus \mathbf{y}) \in N(G)$ as well and, by considering all pairs of addresses, it follows that $N(H) \subseteq N(G)$.

The set of hash functions is a lattice where the functions are ordered by the set refinement relation. The lattice has a smallest element, namely the function that maps every address in main memory to itself, and it has a largest element, namely the function that maps all addresses to the same set (i.e., the one used in a fully-associative cache).

Every function is a refinement of the largest element and is itself refined by the smallest element. The set refinement relation does not order every pair of functions. This happens in particular for the functions used in a skewed-associative cache. The situation can be understood from a fictitious lattice (Figure 1(a)). Each node in the lattice corresponds to a function and the arrows show where the set refinement relation holds. For every function, there is a path from the smallest element to the largest element, passing through that function. The functions labeled G and H are not directly comparable to each other, as there is no path that passes through both G and H. We can however express their similarity by quantifying how much the paths from smallest to largest element for G and H overlap. In the graph, the paths diverge at the hash function I and converge again at S. These functions are the *infinum* (greatest lower bound), respectively *supremum* (least upper bound). Two hash

functions are equal when their supremum equals their infinum (i.e., the paths from smallest to largest element do not diverge at all). The functions are as different as possible when the infinum equals the smallest element and/or the supremum equals the largest element.

2.3. Inter-Bank Dispersion

The supremum hash function and its relation to inter-bank dispersion is illustrated for two set index functions, taken from a family of functions defined in [3]. The index functions H_1 and H_2 are defined by

$$[\mathbf{b}_3, \mathbf{b}_2, \mathbf{b}_1, \mathbf{b}_0]H_1 = [\mathbf{b}_1 \oplus \mathbf{b}_3, \mathbf{b}_0 \oplus \mathbf{b}_2]$$
$$[\mathbf{b}_3, \mathbf{b}_2, \mathbf{b}_1, \mathbf{b}_0]H_2 = [\mathbf{b}_0 \oplus \mathbf{b}_3, \mathbf{b}_1 \oplus \mathbf{b}_2]$$

Every address in main memory is mapped to a set in bank 1 by H_1 and to a set in bank 2 by H_2. These mappings are illustrated in a 2-dimensional plot (Figure 1(b)). Each axis is labeled with the possible set indices for that bank. Every address is displayed in the grid in a position that corresponds to its set indices in each bank. The part x in the address bears no relevance to the value of the index functions.

Both the addresses $x0000$ and $x0101$ map to set 00 in bank 1. The addresses are dispersed in bank 2: $x0000$ maps to set 00 and $x0101$ maps to set 11. Inter-bank dispersion is limited to 2 of the 4 sets: there are no blocks that map to set 00 in bank 1 and either set 01 or 10 in bank 2. This is a consequence of the similarity of the functions H_1 and H_2 and it is described mathematically by the supremum $sup(H_1, H_2)$. The supremum is an imaginary hash function that places the addresses in imaginary sets. In the example, the supremum maps addresses to one of two sets and is defined by:

$$[\mathbf{b}_3, \mathbf{b}_2, \mathbf{b}_1, \mathbf{b}_0]sup(H_1, H_2) = [\mathbf{b}_0 \oplus \mathbf{b}_1 \oplus \mathbf{b}_2 \oplus \mathbf{b}_3]$$

i.e., all addresses with an even number of ones are mapped to one set and those with an odd number of ones are mapped to another set. The supremum is, by definition, refined by both H_1 and H_2. This is shown graphically in Figure 1(b). Set 0 of the supremum corresponds to the upper left-hand square. When refined by H_1, it falls apart into sets 00 and 11 in bank 1. When it is refined by H_2, it splits into sets 01 and 10 in bank 2.

Inter-bank dispersion is always limited to one set of the supremum function. We define the degree of inter-bank dispersion as the 2-logarithm of the number of sets in a bank that have their addresses mapped to the same set of the supremum. It is limited to the range 0 to m.

Definition 2 *The degree of inter-bank dispersion (IBD) equals* $dim\ N(sup(H_1, H_2)) - dim\ N(H_1)$.

It is assumed here that H_1 and H_2 have the same dimensions. For a-way skewed-associative caches ($a > 2$), inter-bank dispersion is defined only for every pair of banks [6]. Using the above definition, one can prove that inter-bank dispersion is maximal ($IBD = m$) if and only if $dim\ N(sup(H_1, H_2)) = n$, i.e., the supremum has only one set. The definition implies an upper bound on inter-bank dispersion: $IBD \leq n - m$, as $dim\ N(sup(H_1, H_2)) \leq n$ and $dim\ N(H_1) = m$. Inter-bank dispersion is limited by the number of hashed address bits n. If n is too small, then the functions H_1 and H_2 will have to be similar to some extent. Maximum inter-bank dispersion can be reached only if $n \geq 2m$.

3. Motivation for Limited Inter-Bank Dispersion

Ideally, a skewed-associative cache should always have as much inter-bank dispersion as possible, because this minimizes conflict misses. There are, however, situations where putting a limitation on inter-bank dispersion is called for. Most processors access the level-1 cache and the TLB in parallel. Therefore, the hash functions operate on the virtual address. In order to avoid aliases in the cache, it is necessary that only *untranslated* address bits (i.e., bits in the page offset) are hashed. Therefore, the page size places a limitation on the number of available address bits n, which in turn limits the inter-bank dispersion ($IDB \leq n - m$).

In some processors the computation of the address partially overlaps the cache access. The cache access starts as soon as the set index bits are computed and the remaining address bits are computed after the cache access starts. By hashing, the number of required address bits rises from m (bit selection) to n. Consequently, the cache delay increases in this type of design.

4. Constructing Hash Functions

Conflict-avoiding hash functions with pre-specified parameters are constructed from profiling information. The method borrows heavily from the method to construct hash functions for direct mapped caches [7, 9].

4.1. Direct Mapped Caches

The estimated number of conflict misses, called the score henceforth, for a hash function is decomposed into a cost for each vector in the null space:

$$score(H) = \sum_{\mathbf{v} \in N(H)} cost(\mathbf{v}).$$

We compute $cost(\mathbf{v})$ as an estimate of the number of conflict misses caused by $\mathbf{v}$, if that vector were a member of the null space of a hash function. The estimates $cost(\mathbf{v})$ for each vector $\mathbf{v}$ are computed during a profiling run of the program, which is explained in detail in [7, 9].

4.2. Skewed-Associative Caches

We estimate the number of conflict misses in a skewed-associative cache as

$$score_{skew}(H_1, H_2) = \frac{1}{3}(score(H_1) + score(H_2) + score(inf(H_1, H_2)))$$

This formula balances the number of conflict misses in each bank (i.e., it tries to minimize the need for inter-bank dispersion) and increments this with the conflicts for the infinum function. Blocks that map to the same set of the infinum function will be mapped to the same set in every bank. Hence, these misses cannot be avoided using inter-bank dispersion and add to the total cost of the hash functions.

We construct a pair of hash functions that minimizes the estimated number of conflict misses for a given benchmark suite by randomly generating 10 million pairs of hash functions as follows. For m columns, n rows and inter-bank dispersion equal to IBD, the two hash functions have $m - IBD$ columns in common. The remaining IBD columns of the hash functions also have to be linearly independent, so a total of $m - IBD + 2 * IBD$ linearly independent columns is needed. For each column, an m-bit random value is generated. If this value is linearly dependent on the previously generated columns, then it is skipped and a new m-bit random value is generated until it succeeds. The constraint on the number of inputs per XOR is enforced by requiring that all randomly generated columns have at most the specified number of 1-bits.

5. Evaluation

The evaluation uses a simulation-based methodology. The SPEC95 benchmarks are simulated using the SimpleScalar tool set. The benchmarks are compiled for the Alpha instruction set using the vendor supplied compiler with optimization flags `-fast -O4 -arch ev6`. In order to match a realistic situation, the profiling is performed by running the benchmarks with the training inputs. The evaluation is performed using the reference inputs. The simulation effort is restricted to representative slices of 500 million instructions per benchmark. The evaluation is performed for an 8-KB data cache with 32-byte blocks. The cache is 2-way skewed-associative, hence there are 128 blocks in each bank and the hash functions compute 7 set index bits.

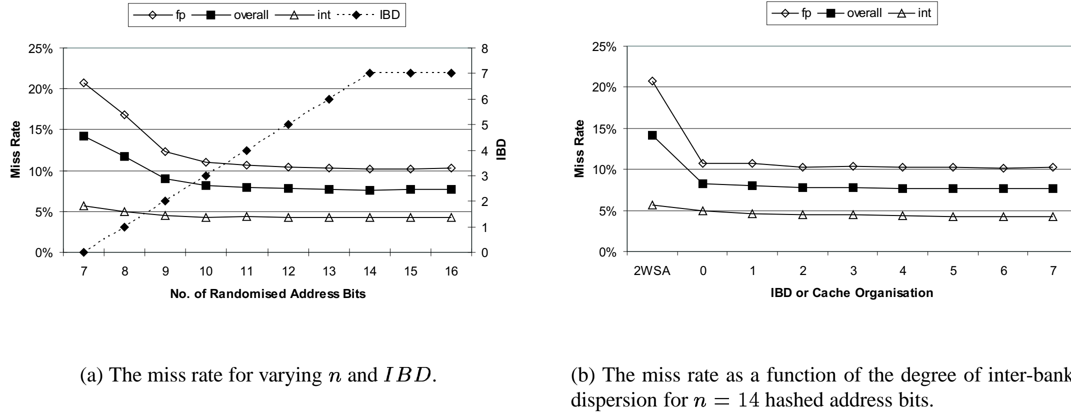
5.1. Number of Address Bits

We analyze the influence of the number of hashed address bits (n) on the miss rate. Inter-bank dispersion is maximum, i.e., $IBD = n - m$. We construct hash functions that minimize the conflict misses for all applications simultaneously. The functions index into an 8-KB 2-way skewed-associative cache with 32-byte blocks ($m = 7$) and have a different number of hashed address bits n. For each value of n, inter-bank dispersion is maximum, i.e., $n - m$. Furthermore, we simulate true LRU replacement in each cache.

Nor maximum inter-bank dispersion, nor a high number of hashed address bits are required (Figure 2(a)). The average miss rate is minimal at 7.9%, while at $n = 10$, a near-optimal miss rate of 8.1% is already obtained. For this configuration, the best degree of inter-bank dispersion that can be obtained is $IBD = 3$.

5.2. Inter-Bank Dispersion

In this experiment, we hash $n = 14$ address bits. Pairs of hash functions are generated that have a degree of inter-bank dispersion that varies from 0 to 7. Figure 2(b) also shows the miss rate in a conventional 2-way set-associative cache (2WSA). The miss rate is almost not affected by the inter-bank dispersion, and decreases only slowly as the IBD is increased. Furthermore, when the IBD is zero, the skewed-associative cache is really a hashed 2-way set-associative cache (in contrast to the conventional 2-way set-associative cache that uses bit selecting mapping). The miss rate of this cache is only slightly higher than the miss rate of the skewed-associative caches. E.g., for $IBD = 7$, the average miss rate is 7.6%, while for $IBD = 0$, the miss rate is



(a) The miss rate for varying n and IBD.

(b) The miss rate as a function of the degree of inter-bank dispersion for $n = 14$ hashed address bits.

Figure 2. The impact of n and IBD on the miss rate.

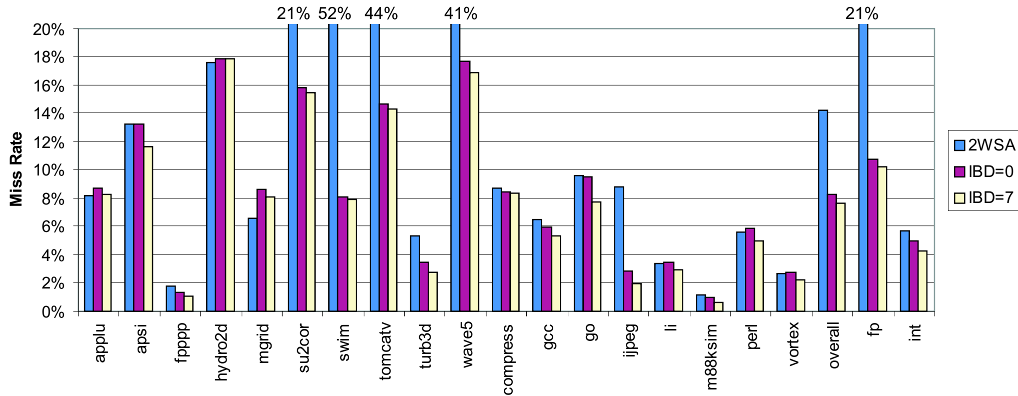


Figure 3. The miss rates for all benchmarks.

8.2%. This indicates that the 2-way skewed-associative cache gets most of its miss rate reduction by using hash functions. The skewed-associative character itself provides a much smaller reduction.

Figure 3 shows the miss rates of the 2-way set-associative cache, the hashed 2-way set-associative cache with $IBD = 0$ and the skewed-associative cache with $IBD = 7$ for each benchmark. We observe different trends for the floating-point and the integer benchmarks. For the floating-point benchmarks, adding hashing to a 2-way set-associative cache has a larger effect on the miss rate than adding skewed-associativity. E.g., for swim, the miss rate decreases from 52% to 8.1% by adding hashing to the set-associative cache. By adding skewing with maximum inter-bank dispersion, the miss rate drops only to 7.9%. There are some exceptions to this rule. For applu, the best hash function is, in fact, bit selection mapping. Therefore, the hash function that we generated introduces more cache misses than the non-hashed cache. By adding skewed-associativity, this negative effect can be countered. A similar explanation holds for apsi, hydro2d and mgrid.

The trend for the integer benchmarks is different. Here, skewed-associativity has a larger influence on the miss rate than hashing. The integer benchmarks have less regular memory access patterns than the floating-point benchmarks, therefore the hash functions (which are static and remain the same during the execution of the program) cannot easily map these patterns in a conflict-free manner in the data cache. Skewed-associativity has the advantage that it can dynamically move data around in the cache using a process called *data reorganization* [3]. This process is useful only when the hash functions cannot already remove most conflict misses.

The conclusion of this experiment is that a better performance/complexity trade-off is made by having little inter-bank dispersion if the intended system will mostly run high-performance applications that typically match the behavior of the floating-point benchmarks. If, however, the system will be used for general-purpose computation and will run programs that are more similar to the integer benchmarks, then it is worthwhile to implement a skewed-associative cache with maximum inter-bank dispersion.

5.3. Complexity of Functions

Finally, we consider the effect of the complexity of the hash functions, which is measured by the number of inputs to each XOR. The number of inputs was unlimited above, but now we limit it to 2. For the floating-point benchmarks, these functions have a slightly, but not significant, higher miss rate.

For the purpose of comparison, the functions proposed in [5] (4 inputs) and those in [3] (2 inputs) are evaluated. Our 4-input functions perform slightly better than the published functions (10.2% vs. 10.5%), while the 2-input functions perform clearly better: 10.4% vs. 12.2%. This validates our approach for generating hash functions and shows that the generated functions are comparable to or better than the best known functions.

6. Conclusion

This paper studies skewed-associative caches and its prominent feature: inter-bank dispersion. We formally define the degree of inter-bank dispersion in a skewed-associative cache and describe the conditions required for maximum inter-bank dispersion.

We demonstrate that applications do not require a high degree of inter-bank dispersion, but a small amount of it is able to remove most misses. We examine trade-offs of the skewed-associative cache with respect to the degree of inter-bank dispersion, the number of hashed address bits and the complexity of the hash functions. A skewed-associative cache with a near-optimal complexity/performance trade-off is proposed: The optimized 8-KB two-way skewed-associative cache maps 11 address bits onto 7 set index bits and has a degree of inter-bank dispersion equal to 3.

There are two components to the miss rate reduction of the two-way skewed-associative cache, namely hashing and inter-bank dispersion. Hashing by itself removes almost all conflict misses in the SPEC95 floating-point benchmarks, while skewing provides little benefit. For the integer benchmarks, hashing and skewing contribute about evenly.

REFERENCES

[1] A. González, M. Valero, N. Topham, and J. Parcerisa, "Eliminating cache conflict misses through XOR-based placement functions," in *ICS'97. Proceedings of the 1997 International Conference on Supercomputing*, pp. 76–83, July 1997.

[2] H. Vandierendonck and K. De Bosschere, "Evaluation of the performance of polynomial set index functions," in *WDDD: Workshop on Duplicating, Deconstructing and Debunking, held in conjunction with the 29th International Symposium on Computer Architecture (ISCA-29)*, pp. 31–41, May 2002.

[3] F. Bodin and A. Seznec, "Skewed associativity enhances performance predictability," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pp. 265–274, June 1995.

[4] A. Seznec, "A case for two-way skewed associative caches," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pp. 169–178, May 1993.

[5] A. Seznec and F. Bodin, "Skewed-associative caches," in *PARLE'93: Parallel Architectures and Programming Languages Europe*, (Munich, Germany), pp. 305–316, June 1993.

[6] H. Vandierendonck and K. De Bosschere, "On null spaces and their application to model randomisation and interleaving in cache memories," Tech. Rep. DG02-02, Ghent University, Dept. of Electronics and Information Systems, July 2002.

[7] H. Vandierendonck and K. De Bosschere, "Efficient profile-based evaluation of randomising set index functions for cache memories," in *2nd International Symposium on Performance Analysis of Systems and Software*, pp. 120–127, Nov. 2001.

[8] M. D. Hill and A. J. Smith, "Evaluating associativity in CPU caches," *IEEE Transactions on Computers*, vol. 38, pp. 1612–1630, Dec. 1989.

[9] H. Vandierendonck and K. De Bosschere, "Highly accurate and efficient evaluation of randomising set index functions," *Journal of Systems Architecture*, vol. 48, pp. 429–452, May 2003.