Invertible Bloom Lookup Table Cache

Parker Lamb

Fourth Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2016

Abstract

This report presents the possibility of implementing a CPU cache utilising an invertible Bloom lookup table, a probabilistic data structure similar to a hash map. Caches make up a significant portion of the die area in modern processors, subsequently any gains in cache efficiency will be also be proportionally large. The invertible Bloom lookup table allows can place a cache block anywhere in the cache, potentially leading to an increase in cache performance as mapping conflicts in the cache are reduced. The new cache design was created in hopes of exploiting the invertible Bloom lookup table's ability to reduce conflicts into improved cache performance.

A design for an invertible Bloom lookup table cache was created, with some adaptations to the invertible Bloom lookup table for use in hardware. A simulator was written with an implementation of the cache to test performance. Varying configurations of the cache were generated and tested with the simulator, using memory traces from a subset of the SPEC2006 benchmark set as inputs.

Characterisation of the cache was performed during analysis; it was discovered that the proportion of the total cache space split between the mapping structure and the storage structure is the most influential parameter, apart from raw cache size. Unfortunately, it appears that the invertible Bloom lookup table cache is not a viable design. The cache performed worse than a comparable set-associative cache in all test cases.

Acknowledgements

I would primarily like to thank Christophe Dubach for endless help and support throughout this project and for providing the original idea.

Thanks also go to William Gardner for thinking outside the box and generally keeping me sane throughout this project.

Contents

1 Introduction **7**
1.1 High Level View . . . 7
1.1.1 Contributions . . . 8
1.2 Report Structure . . . 8

2 Background **9**
2.1 Memory Hierarchy . . . 9
2.2 Classical Cache Structure . . . 9
2.2.1 Set-Associative Caches . . . 10
2.2.2 Fully Associative Caches . . . 11
2.2.3 Direct Mapped Caches . . . 11
2.3 Cache Miss Types . . . 12
2.4 Cache Write Policies . . . 12
2.5 Bloom Filters . . . 13
2.6 Invertible Bloom Lookup Table . . . 14
2.6.1 Insertion Operations . . . 15
2.6.2 Lookup Operations . . . 15
2.6.3 Deletion Operations . . . 16
2.6.4 Time Complexities . . . 16

3 Related Work **17**
3.1 Introduction . . . 17
3.2 Cuckoo Hashing . . . 17
3.2.1 Lookups . . . 18
3.2.2 Insertions . . . 18
3.2.3 Deletions . . . 19
3.3 Skewed-Associative Caches . . . 19
3.4 Summary . . . 19

4 Invertible Bloom Lookup Table Cache **21**
4.1 Introduction . . . 21
4.2 Structure . . . 21
4.2.1 Lookup Operations . . . 22
4.2.2 Insertion Operations . . . 23
4.2.3 Deletion Operations . . . 23
4.2.4 Cache Access Overview . . . 23

4.3 Design Restrictions 24
4.4 Optimisations 25
4.4.1 IBLT Lookup Operation Optimisation 25
4.4.2 IBLT Insertion Operation Optimisation 25
4.4.3 IBLT Cell Field Size Reduction 26
4.5 Parameters 26

5 Cache Simulator 29
5.1 Overview 29
5.2 Custom Data Structures 30
5.3 Set- and Fully-Associative Cache Implementations 31
5.3.1 Statistics Tracking 31
5.3.2 IBLT Cache Implementation 32
5.3.3 Hash Functions 33
5.3.4 Statistics Tracking 34

6 Methodology 35
6.1 Introduction 35
6.1.1 Dynamic Binary Instrumentation 35
6.2 Benchmarks 36
6.3 Basic Block Capture 36
6.4 SimPoint 37
6.4.1 Background 37
6.4.2 Usage 39
6.5 Memory Trace Capture 39
6.6 Simulations 40

7 Evaluation 41
7.1 Design Space 41
7.2 Samples 42
7.3 Results 43
7.3.1 Invertible Bloom Lookup Table Cache Characterisation 43
7.3.2 Comparison of Classic and IBLT Caches 52
7.3.3 Summary 53

8 Conclusion 57
8.1 Future Work 57
8.2 Critical Analysis 57
8.3 Summary 58

Bibliography 59

Chapter 1

Introduction

The following report explores the possibility of implementing a hardware memory cache using an invertible Bloom lookup table, a data structure which utilises hashing to maintain a set of mappings. Caches have become increasingly large in recent times and take up significant area on processor dies, occupying approximately 50% of the total space in some modern designs [6]. As such, optimisation of caches is desirable due to the proportional increase in efficiency. If caches can be shrunk, or performance improved within the same or smaller chip area, the processor will become more efficient due to better utilisation of resources. Recent trends in processor development have been towards making processors increasingly power efficient due to the increases in transistor count and the amount of heat which they generate [6]. This research was undertaken in hope of contributing to this effort.

1.1 High Level View

The idea presented in this report is a new cache design, called an "invertible Bloom lookup table cache", which attempts to overcome some of the limitations of existing caches. One of the problems commonly encountered in designing caches is the difficulty of effectively mapping cache blocks to locations in the cache. Nearly all commonly used mapping schemes require blocks to be evicted in the event of mapping conflicts, as discussed in Section 2.1, which this design aims to reduce. While blocks are typically evicted in least-recently-used order (itself an approximation of longest-time-to-next-use, the optimal block eviction policy[1]), this is not always possible and still creates unnecessary overhead. The new cache architecture showed here is designed to reduce the amount of conflicts in the cache, thus improving performance.

The invertible Bloom lookup table cache uses a separate data structure (the invertible Bloom lookup table) to maintain the mappings between cache block identifiers and

[1] Longest-time-to-next-use can be intuitively seen as the best block eviction policy; it will keep the most commonly used blocks in the cache for the next step in execution. Least-recently-used is implemented in practice because it is impossible to determine which block will be used furthest in the future.

the locations of their respective blocks in the cache. Cache blocks themselves can be placed anywhere in the cache, evictions caused directly by the cache only occur when there is insufficient space to hold incoming blocks. Conflicts can still happen within the mapping structure, though these can be guaranteed to occur at a known rate with proper configuration.

1.1.1 Contributions

The primary object of this project was to research the possibility of a hardware implementation of an invertible Bloom lookup table, a software data structure, in the context of a cache. A suitable implementation of this cache was designed and optimised. The various configuration parameters of the cache were explored through a number of simulations. Performance data of various types were collected from the simulations, from which analysis on the cache's suitability was conducted.

1.2 Report Structure

This report is split into seven chapters, not including this introduction. Chapter 2 describes relevant background information for the remainder of the report, including existing cache architectures and the original research on invertible Bloom lookup tables. Chapter 3 discusses related work, notably the use of hashing in caches. Chapter 4 details the invertible Bloom lookup table cache architecture, along with some design restrictions and optimisations. In Chapter 5 the implementation of the cache simulator which was used to collect performance data is described. Chapter 6 discusses the methodology for data collection used in the project, which involves several steps and tools. Chapter 7 concerns the evaluation of the data generated by the cache simulator, and includes a discussion of the design space and how samples were taken. Chapter 8 concludes the report with a summary of the project and its results, a critical analysis of the work conducted during the project, and potential future work.

Chapter 2

Background

2.1 Memory Hierarchy

Computer memory is organized into a hierarchy, with smaller, lower-latency memories at the top of the hierarchy and larger, higher-latency memories at the bottom [6]. Memory hierarchies are used due to the extremely high latency of large-capacity memories with current technology [6]. Subsets of the larger memories can be loaded into the smaller memories for quicker access [6].

Caches are part of the memory hierarchy, occupying the top two or three layers in a modern computer [6]. An example memory hierarchy is shown in Figure 2.1 (adapted from [6]). The area occupied by each memory in the figure denotes relative size, though the representation shown is not proportional. Memory access latency increases moving towards the bottom of the hierarchy (hard drive). L1 and L2 refer to the first and second level caches, respectively. The caches discussed in this report, unless explicitly stated otherwise, are considered to be L1 data caches.

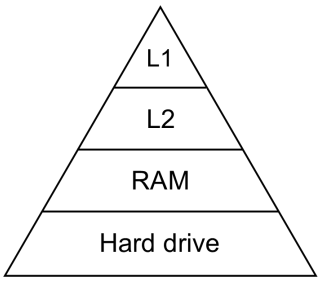

Figure 2.1: Example Memory Hierarchy

2.2 Classical Cache Structure

There are three common types of cache architecture: set-associative, fully associative, and direct mapped [6]. Set-associative and direct mapped caches are the most common

found in today's processors, whereas fully associative caches have limitations which prevent them from being commonly used [6].

2.2.1 Set-Associative Caches

Set-associative caches are divided into a collection of small structures called sets [6]. Sets have a fixed number of ways, which are slots where cache blocks can be placed [6]. Blocks are inserted into a specific set selected by the addressing logic in the cache [6]. The cache uses the memory address of an access to locate and insert cache blocks [6]. Figure 2.2 shows a simplified view of a set-associative cache. Each set has two ways, with darker colouring indicating that the way is occupied.

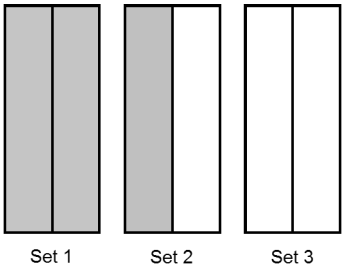

Figure 2.2: Set-Associative Cache Layout

Addresses are broken down into several parts by the cache hardware in order to select where the block should be placed. Figure 2.3 shows the corresponding breakdown for a set-associative cache with a 64-byte block size and 128 sets, using 64 bit addresses. The numbers seen in the figure correspond to bit offsets in the address. Offsets may be in different locations from those depicted here depending on cache architecture, block size, and address length.

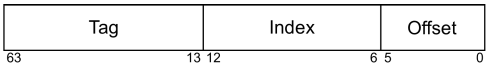

Figure 2.3: Breakdown of a Cache Block Address

The bottom six bits are the offset in the cache block where the access takes place. These are not used by the cache because blocks are multiple bytes, meaning that any requested data will be part of a larger block and irrelevant to the cache as it operates at a block-granularity. All accesses for which the bits in positions 6-63 are the same will occur in the same cache block. Offset sizes are calculated by the following equation (adapted from [6]):

$$\textit{offset size in bits} = \log_2 (\textit{block size in bytes})$$

The index field is calculated using similar logic to the offset field [6]. The same base-2 logarithm is used, except with the number of sets in the cache as the input. This will produce the number of bits required to access the cache set with the largest index.

Sets are accessed in a similar fashion to arrays, where the nth element is at offset n-1, assuming $n > 0$. The value found in the index field is used to select the cache set to be checked for the requested block.

The tag field is made up of the remaining bits in the address, however many there are [6]. Tags are used as identifiers for cache blocks within sets. When the cache is searching for a block in a set, the tag field is used for comparison.

Once the set is selected, a cache block can be placed in any empty slot in the set [6]. If all slots are occupied, a block is evicted, usually based on whichever block was least recently accessed [6]. For example, if a block were to be inserted into set 1 in Figure 2.2, one of the blocks already resident would need to be evicted beforehand. Set 2, on the other hand, has an empty space so an insertion could take place there without requiring an eviction.

Blocks are retrieved in much the same way they are inserted. The correct set is found using the same indexing process, though all ways in the set must be searched individually to find the correct block because there is no internal organisation scheme for the placement of cache blocks within a set [6].

2.2.2 Fully Associative Caches

Fully associative caches offer the best theoretical performance of any cache. They do not suffer from conflict misses, which eliminates an entire category of misses [6]. Fully associative caches are most easily viewed as a special case of set-associative where there is only a single set and the number of ways is equal to the number of blocks in the cache. As such, any block can be placed anywhere in the cache with no conflicts. Unfortunately, it is not possible to implement fully associative caches in hardware because the lookup logic is too time-consuming [6]. Each block in the cache must be checked individually during lookups which takes far longer than other caches' block placement schemes which use indexing to narrow down the search when locating blocks.

2.2.3 Direct Mapped Caches

Direct mapped caches are the simplest of the three types presented here and also tend to provide the worst performance [6]. Direct mapping refers to the block placement scheme used by this cache, in which a given block can only be placed at a single specific location in the cache [6]. Similarly to fully associative caches, direct mapped caches can be conceptually seen as a set-associative cache with only a single way per set. This cache design can lead to problems with excessive block eviction due to the issues with conflicting blocks repeatedly evicting one another. Direct mapped caches tend to have much higher conflict miss rates than set-associative caches with multiple ways [6].

2.3 Cache Miss Types

Cache misses are broken down into three distinct types, colloquially referred to as the "Three Cs": compulsory (also known as cold), conflict, and capacity [6]. Tracking cache misses in general is important because a count of cache misses is required for calculating the miss rate of the cache, the most commonly quoted metric for performance. Cache miss rate is calculated by dividing the number of misses by the total number of cache accesses

Cold misses are those which are unavoidable because the relevant block has never been accessed and never has had a chance to be inserted into the cache [6]. If a theoretical infinite cache were to exist, cold misses would still occur, intuitively one for each cache block in a given program [6].

Capacity misses occur when there is no space left in the cache for an additional block [6]. Misses of this type result in a conflicting block being evicted from the cache so a new one can take its place [6]. All caches suffer from capacity misses, as a cache must be of finite size.

Conflict misses are a result of a too many blocks occupying a single set in a set-associative or direct mapped cache [6]. Most caches only allow blocks to be placed in certain areas of the cache due to the underlying indexing logic used to determine where to locate cache blocks [6]. This allows for quick access times, though at the cost of evicting blocks when two conflict [6]. Conflict misses are only seen in set-associative and direct-mapped caches, they are not seen in fully-associative caches as blocks can be placed anywhere in the latter [6].

2.4 Cache Write Policies

A cache write policy is the logic governing a cache's behavior when performing writes to memory [6]. Caches use one of two write policies; write-back or write-through [6]. A write-back cache only writes to cache blocks at the highest level of the hierarchy (the first level cache) [6]. Changes to modified cache blocks are only propagated to lower memory levels when those blocks are evicted [6]. This method reduces write latency in the cache as only one level of the hierarchy is modified during a write operation. Write through caches propagate changes to all levels of the hierarchy during writes [6]. This write policy simplifies the eviction and update logic at the expense of a longer write time [6].

There are two accompanying policies for the action to be taken on a write miss; when a cache tries to write to a block that is not resident. These are write-allocate and write-no-allocate [6]. Write-allocate caches load the relevant block into the cache in the event of a write miss [6]. Write-no-allocate caches do not load missing blocks in this instance and simply perform the write at a lower level in the cache [6].

The write-back and write-allocate policies are naturally grouped together [6]. It does not make sense for a cache to load blocks and then not write to them and vice versa.

The write-through and write-no-allocate policies are paired for similar reasons [6]

The caches used in this project use write-back and write-allocate policies. Write-allocate was chosen as a write-miss policy because it stresses the insertion and eviction logic of a cache more than write-no-allocate due to the increased number of blocks which must be brought into the cache. This was desirable when testing the performance of a cache. Write-back was chosen as a write policy because of its natural compatibility with write-allocate.

2.5 Bloom Filters

A Bloom filter is a probabilistic data structure which offers a trade-off between required storage space and access time, at the cost of less than 100% accuracy in response correctness [1]. Data cannot be retrieved from a Bloom filter, it can only be inserted and then checked if resident. Bloom filters are best suited to applications where accessing data is an expensive operation which can be optimised by eliminating superfluous accesses [1]. A Bloom filter is well suited to this task because it does not return false negatives (a true response when a false one would be correct) and is quick to access [1]. Bloom filters are a key part of invertible Bloom lookup tables, the core component of this project. The relationship between Bloom filters and invertible Bloom lookup tables is detailed in Section 2.6.

A Bloom filter can be conceptualised as a string of bits of arbitrary length and an accompanying hash function, the output size of which is identical to the bit-string length [1]. The hash function itself can take any form, though one which distributes its inputs as evenly as possible in the output space is desirable [1]. The bits in the filter are all initialised to zero when the structure is empty [1].

Data is inserted into the filter through use of the hash function [1]. Each piece of data is hashed by the function and the resulting bit-string is inserted into the table using the bitwise-or function [1]. The filter will slowly fill up with 1's as data is inserted.

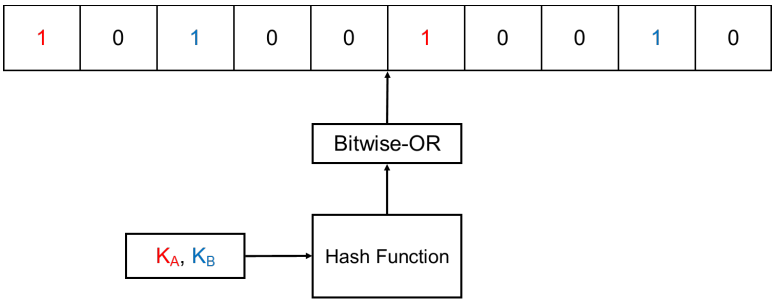

Figure 2.4: Bloom Filter Insertion Example

Figure 2.4 shows an example of this insertion process. Two pieces of data, K_A and K_B, are hashed by the hash function. They are added to the Bloom filter using the bitwise-or function, with the resulting 1's denoted by arrows. The bits in the table which were

changed to 1's are colour-coded to show which piece of data they represent. K_A's hash has 1's in the 1st and 6th bits, while K_B's has 1's in the 3rd and 9th bits.

Checking if a piece of data has been inserted into the table is done in a similar fashion to insertion. The new piece of data is hashed and then compared with existing data in the filter [1]. If the filter contains a zero in any of the positions where the hashed data has a one, then it can be determined that the data has never been inserted into the table [1]. False negatives will never be produced as there is no way for a bit to be reset to zero once it has been set to one, ensuring total accuracy in this sense. This is a useful property because the filter will only cause extra accesses to an underlying data store, never prevent them.

Bloom filters suffer from a phenomenon known as aliasing, where insertions of other pieces of data can conflict with the checking of the desired piece [1]. If enough data is inserted into the filter, it is possible for the filter to return a false positive – a true response where there should have been a false one. False positives can be counteracted to some degree by including more bits in the filter.

2.6 Invertible Bloom Lookup Table

An invertible Bloom lookup table (IBLT) is a data structure that utilises Bloom filters to store key-value pairs in a hash table-like structure, as described by Goodrich and Mitzenmacher [4]. Unlike a standard Bloom filter, it allows for the retrieval of data rather than only testing if it has been previously inserted [4]. The interface for an IBLT is similar to a standard hash table, with functions for insertion, retrieval, and deletion of key-value pairs.

Internally, an IBLT consists of a table made up of a number of cells and a set of hash functions [4]. Each cell has several data fields which are used to store the key-value pairs and associated metadata [4]. The hash functions are used to map each input key to cells in the table [4]. These hash functions are assumed to be random and have outputs which are equally distributed across the output space [4]. Each hash function has an equal partition of the table from which it selects cells [4]. This prevents hash collisions between the functions, eliminating a potential source of error [4].

The data fields contained in a cell can vary, though the standard structure has three: a field for keys, a field for values, and a field which stores the number of key-value pairs that are contained in the cell [4]. The key and value fields in the cell are treated as Bloom filters; when data is inserted or removed from these fields the new data is added to the existing data in the field using the exclusive-or function [4]. The counter field is an integer which is incremented and decremented during insertion and deletion operations, respectively [4]. These fields can be configured to have specific numbers of bits depending on what the use case for the IBLT requires [4].

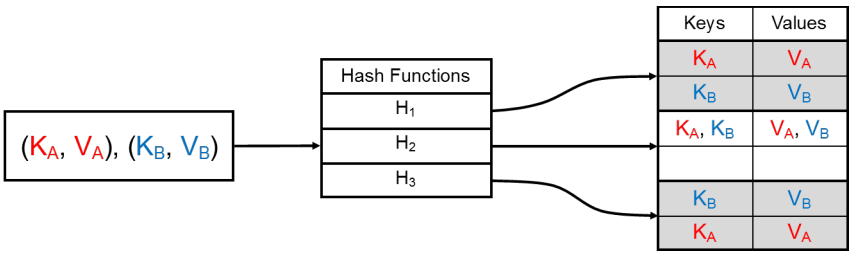

Figure 2.5: IBLT Insertion Example

2.6.1 Insertion Operations

During an insert operation, each hash function is executed on the given key and the selected cells are updated with the new key-value pair while the count field in incremented [4]. Every key-value pair inserted into the IBLT will be inserted into the underlying table once for every hash function. Figure 2.5 shows this process in greater detail.

In the diagram, two key-value pairs, (K_A, V_A) and (K_B, V_B), are inserted into the IBLT. There are three hash functions, $H_1 - H_3$, each with its own two-element partition in the table. The partitions are coloured differently for readability. H_1 and H_3 hash both keys without collisions, these are inserted into separate cells within the respective partitions, though at different locations within the cell. H_2 hashes both keys to the same cell in the partition, leaving the other empty. The count fields for all cells have been incremented accordingly.

2.6.2 Lookup Operations

Lookup operations work very similarly to insertion operations. The same hashing process is used to locate the correct cells in the table, after which the key field in each cell is checked against the desired key passed to the retrieval function [4]. If a match is found, the value field of the matching cell is returned as the result of the lookup operation [4]. A value will only be returned from the lookup operation if there exists a cell in the table to which the only the given key and no others have been hashed [4].

Due to the inherent risk of hash collisions, an IBLT does not provide a guarantee that an inserted key-value pair can always be retrieved. The probability of successfully retrieving a key-value pair can be calculated with the following equation:

$$p_0 = \left(1 - \frac{k}{m}\right)^{(n-1)} \approx e^{\frac{-km}{n}}$$

where p_0 is the probability of success, k is the number of hash functions in the IBLT, m is the total number of cells in the IBLT, and n is the number of keys which have been inserted into the IBLT [4].

This equation gives a guarantee that a key-value pair can be retrieved with a certain probability, a useful property of the data structure [4]. The parameters of the structure to be tweaked to obtain a desired performance rate [4].

2.6.3 Deletion Operations

Deletion of key-value pairs from an IBLT is done almost identically to insertion operations. The key is hashed and used to select cells from the table [4]. The given key-value pair is removed from key and value fields in the selected cells with the use of the exclusive-or function [4]. The exclusive-or function is reversible, which is why it is used for combining values in the IBLT. This can be seen in the following equation:

$$(A \oplus B \oplus C) \oplus (A \oplus B) = C$$

After the key and value have been removed from the cell, the count field is decremented to account for the lost data [4].

Deletion operations in an IBLT are unsafe. If a deletion is performed with data that has not been previously inserted, the operation will still "succeed" and the data in the IBLT will be corrupted [4]. Because exclusive-or is reversible, a deletion with previously un-inserted data will have the effects of an insertion, except that the count field in the cells will be decremented instead of incremented. This can lead to lookup operations succeeding where they should fail.

2.6.4 Time Complexities

It can be intuitively seen that the insertion, lookup, and deletion operations all have the time complexity $\theta(1)$. All operations are bounded by the number of hash functions in the IBLT, as all operations iterate over the hash functions to run. The number of inputs given to the IBLT has no effect on its logic, meaning that all operations can be guaranteed to complete in constant time.

Chapter 3

Related Work

3.1 Introduction

The area of hashing has long been a field of research in computer science. Some data structures which utilise hashing, like hash maps, are now ubiquitous in modern programming languages. The aim of this chapter is to explore some of the existing work in the field concerning the use of hashing in data storage. Cuckoo hashing, an approach used to resolve hash collisions between data elements in a key-value data structure, is discussed, as well as skewed-associative caches, which are an implementation of Cuckoo hashing in a CPU cache. Some similarities may be noted between these approaches and the one used in the IBLT cache idea presented in this report.

3.2 Cuckoo Hashing

Cuckoo hashing was originally described by Pagh and Rodler [8]. Their work outlines the design for a hash table structure which can dynamically resolve hash collisions as they occur [8]. This is in contrast to a typical hash table where elements that have the same computed hash are placed into "buckets" which must be searched sequentially if they contain more than one element [3]. A standard hash table will have O(n) worst case lookup time because of this searching, though the amortised average case is usually O(1) [3]. Cuckoo hashing offers O(1) lookup time in the worst case and the amortised average case, meaning that a Cuckoo-based hash map has the potential to be much more efficient than a standard one [8]. There are downsides to Cuckoo hashing, notably when inserting data into the structure, as many re-hashings may be required before the data can be successfully inserted [8].

Cuckoo hashing works by partitioning a hash table into two parts, each with its own respective hash function [8], similar to how the partitioning of cells works in an IBLT. The design outlined in the original paper suggests that the hash functions be implemented in parallel if there is hardware support to improve performance [8]. The hashing process uses the two tables in tandem during insertions, deletions, and lookups

[8].

3.2.1 Lookups

Lookups can be done in O(1) time in all cases [8]. Both hash functions are evaluated with the given key and the data value in the corresponding hash table location is returned, provided it is not null [8]. A Cuckoo-hash table defines a particular value which denotes "null," typically zero [8]. Null values are not returned and considered a lookup failure in their respective tables [8]. If failures occur in both tables, the entire lookup operation fails [8].

3.2.2 Insertions

Insertions are the most complicated operation performed by a Cuckoo-hash table. First, a lookup operation is performed to see if the data has already been inserted into the table [8]. If it has, the insertion operation immediately completes as it is now redundant [4]. Otherwise the input data is hashed using the first hash function and the corresponding table is checked to see if the data can be placed in the location indicated by the hash result [8]. If this succeeds, the insertion operation terminates as it has successfully inserted the data into the table [8]. If the data cannot be placed in the table because other data is resident at that location, a process of moving keys begins in order to free up the slot [8].

Resolving hash collisions is done by swapping keys between locations in the table. The conflicting key in the table found during an insertion is swapped with the key corresponding to the new data [8]. The key for the new data is now resident in the table. The evicted key must be re-inserted into the table; this is done by hashing it using the second hash function in the structure [8]. If the key can be inserted into the second table, then the insertion completes as there are no more collisions [8]. Otherwise, the process continues in a loop until it finishes or reaches the maximum number of iterations [8].

The tables are re-hashed with a new pair of hash functions if the maximum number of iterations is reached [8]. The maximum number of iterations, referred to as "MaxLoop" in the original research, is defined as:

$$MaxLoop = \lceil 3 \log_{1+\varepsilon} r \rceil$$

where ε is a constant > 0 such that

$$r \geq (1+\varepsilon)n$$

n being the number of keys in the hash table and r being is the number of memory words in a single hash table [8].

If a re-hash occurs, all existing keys in the structure are examined and removed if they are in the incorrect position and re-inserted after being hashed with the new functions [8]. A re-hash of the data is sometimes accompanied by a resize of the hash tables themselves, the size of which is increased if the number of occupied cells in either is one more than half of the total [4]. A table re-hash is also forced every n^2 insertions, where n is the size of the table [8].

3.2.3 Deletions

Deletions are handled in a similar fashion to lookups, and share the same time complexity [8]. Both hash functions are again computed with the given key to be find the relevant locations in the tables [8]. The located values are deleted by setting them equal to the null value specified by the structure [8].

3.3 Skewed-Associative Caches

A skewed-associative caches is conceptually quite close to a set-associative cache, though with a slightly more clever hashing scheme . They feature a similar idea to that used in cuckoo hashing and IBLTs; utilising multiple hash functions to distribute data into distinct sets.

The skewed-associative cache outlined by Seznec [9] has two sets where cache blocks are stored. Two hash functions are used to insert blocks into the cache [9], much like Cuckoo hashing. The benefit of this approach is that conflicts between blocks in one set can be reduced or eliminated in the other set, though this is dependent on the implementation of the hash functions [9]. The hash functions are not designed necessarily to distribute cache blocks evenly into the output space, merely to adjust the computed indices enough to remove some of the block collisions [9].

3.4 Summary

Previous approaches to space-efficient hashing have been primarily focused on resolving hash collisions by re-hashing. Re-hashing can be effective, though in worst-case scenarios it requires a significant amount of effort to insert keys, as seen above. Re-hashing is better suited to situations where there is not sufficient space to maintain an expandable hash map-like structure which can simply acquire more memory when necessary. An IBLT is something of a midpoint between the two approaches. It has significant space for storage but is not dynamically expandable and does not re-hash data. The lookup, insertion, and retrieval operations in an IBLT all have time complexity $\theta(1)$, an advantage over all of the other hashing structures discussed here.

Chapter 4

Invertible Bloom Lookup Table Cache

4.1 Introduction

The invertible Bloom lookup table cache is an attempt to combine some of the better properties of hash-based data structures with those of caches. The ultimate aim is to produce a cache which can store cache blocks with minimal conflicts while using approximately the same hardware resources of an existing cache of similar size.

4.2 Structure

The IBLT cache is utilises an invertible Bloom lookup table to store mappings and a backing array to hold cache blocks as the underlying storage structure. Cache block addresses are used as identifiers for the cache blocks, similar to tags in a traditional cache. These addresses are used as inputs into the hash functions of the IBLT during insertion, lookup, and deletion operations.

The IBLT stores mappings between memory addresses and indices in the backing array of cache blocks. When blocks are inserted into the cache, the block data is stored in the array and a key-value pair of the block's address and the corresponding array index is stored in the IBLT. The array of cache blocks also contains storage for the raw (unhashed) memory address of the block along with the block data. The raw address is required during deletion operations in the IBLT; an exact mapping between the address and the block data must be maintained for this reason. The actual structure used for the backing array of cache data must support direct indexing and deletion of elements and maintain block order by least-recently-used (LRU). A linked doubly-linked list was used in the implementation of the simulator.

The hash functions used by the IBLT are not explicitly defined, as in the original research. The functions used in the implementation of the simulator are selected randomly from the universal family H_1. This is discussed in detail in Section 5.3.3.

Figure 4.1 shows a high-level view of the IBLT cache. This cache has an IBLT with

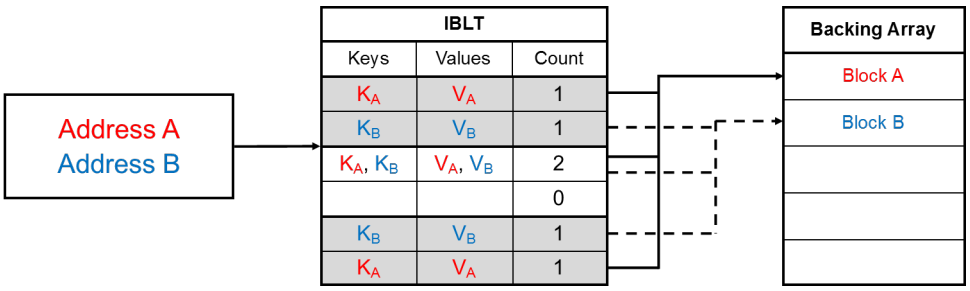

Figure 4.1: IBLT Cache Layout

three hash functions and partitions, darker colouring shows the grouping of the partitions. The hash functions are not shown as their functionality is internal to the IBLT. The cache's backing array has space for five cache blocks. Some of the connections between the IBLT the backing array are dashed for readability, there is no difference in functionality. Cache blocks at addresses A and B have previously been inserted into the cache, all corresponding values are differentiated by colour coding.

The data stored in the IBLT are the mappings between memory addresses and cache blocks. The keys (K_A and K_B) are the addresses A and B, respectively. The values (V_A and V_B) are the indices of the corresponding cache block in the backing array. Because there are multiple insertions of each key-value pair, there are multiple references to the same cache block, denoted by lines between the IBLT and the backing array. A cache block can be retrieved as long as there is a single usable reference in the IBLT.

This cache arrangement aims to duplicate the retrieval performance of a fully associative cache and the fast lookup time of a set-associative cache. The need to check every tag in the cache during lookups as with a fully associative cache is eliminated. The ability to place a cache block anywhere in the cache is retained, giving an advantage over set-associative caches which must evict cache blocks when there are too many in a single set.

4.2.1 Lookup Operations

Cache lookups are performed by hashing the input address of the desired memory block with each hash function in the IBLT, successively. If a cell is found where only a single matching address has been hashed, then the corresponding index value is returned and the memory can be directly accessed in the cache block array. In this instance the operation is classed as a cache hit. In any other case the cache block may not be retrievable. If none of the cells checked during the hashing process contain the desired address, then the operation results in a cold miss or a conflict miss, depending on whether the block in question has been previously resident in the cache. Similarly, if all of the cells contain more than one entry then the cache block cannot be retrieved because it is not possible to ensure that only bits belonging to the value are read from the cell. A conflict miss will occur in this case. Like a traditional cache, lookups are not guaranteed to succeed.

4.2.2 Insertion Operations

Insertions into the cache are handled by hashing the incoming block's address with every hash function in the IBLT. The count field is checked in each cell selected by the hash functions to ensure that the limit of inserted key-value pairs is not being exceeded. If each cell has space for a new entry, the new address is added to the cell along with the index of the cache block in the backing array. This is done through exclusive-or operations are performed with the elements the new key-value pair and the existing key-value data, after which the existing data is overwritten with the results. The count field is also incremented when the data is inserted. The cache block is not inserted into the backing array until it is ensured that the corresponding mapping can be held in the IBLT. If one or more cells do not have room for the additional data, then the least-recently-used cache block is removed from the cache and the insertion is attempted again. This sequence continues until the insertion succeeds. The same process is used if there is no space remaining for the block in the backing array.

4.2.3 Deletion Operations

Deletions also require the block's address to be hashed with every hash function to ensure that all entries have been removed. Careful reasoning is required for safe deletions due to the fact that the deletion logic results in undefined behaviour when there is an attempt to delete an entry which has not been inserted into the table. The exclusive-or function which is used for combining data in cells is reversible only when previously seen values are used as input. For this reason, deletions are done using address data from the backing array or cells with single entries as it is guaranteed that this data has previously been inserted into the IBLT. The deletion involves an exclusive-or operation with the existing key-value data and the key-value pair given as an argument to the function. The data remaining in the cell after the operation will no longer contain the removed key-value pair. The count field is also decremented.

4.2.4 Cache Access Overview

First, when the cache is accessed, a lookup operation is performed to determine whether the requested block is resident in the cache. If it succeeds, the relevant data is simply returned. Otherwise the block must be inserted into the cache as per the write-allocate policy. If this occurs, then there must be at least one empty space in the cache block array to hold the new block. If no spaces are available after a failed lookup operation has occurred, the least-recently-used block is evicted from the array and the corresponding mapping deleted from the IBLT.

A lookup operation can fail with one of two results. Either the cache block is determined to not currently be resident, or that it is impossible to decide whether it is resident in the cache. If there is a cell found during the lookup operation which contains zero or one (non-matching) entries, it can be guaranteed that the cache block is not in the cache. The block can be immediately inserted in this case because there

is no risk of duplicating a block already resident in the cache. Otherwise it cannot be determined whether the requested block is currently in the cache. In this case the cache is flushed and the block can be safely inserted.

4.3 Design Restrictions

There are several limitations placed on the IBLT cache for practical reasons. The number of key-value pairs allowed to be stored per IBLT cell is artificially restricted and the insertion and deletion logic of cache entries is somewhat governed by properties of the IBLT.

Each cell in the IBLT can in theory contain an unlimited amount of data, all of which will be compressed to the size of the key and value fields using the exclusive-or function. This compression is lossy in some sense due to the fact that the only one of the compressed values can be reconstructed at a time, and only if all of the other uncompressed values are available to the hashing process. This reversibility property is illustrated in the following equation:

$$(A \oplus B \oplus C) \oplus (A \oplus B) = C$$

Any element of data can be fully retrieved from the cell though this requires expensive copy operations if the operation is to be non-destructive. All other data except the desired value must be removed from the cell for a successful recovery. Because the IBLT cache is designed to be implemented in hardware, it is infeasible to perform such operations as massive duplication of storage to hold copies of the structure would be required. As such, the number of key-value pairs which can be inserted into an individual cell has been artificially limited to reduce the amount of work required to operate on the IBLT when hash collisions occur. The upper limit of entries is defined by a configurable parameter in the simulator.

In some instances, the cache must be flushed before new data can be inserted. This is dependent on the state of the IBLT when the insertion is attempted. If it is not possible to determine whether a block is currently resident in the cache, then action must be taken to ensure that a duplicate entry is not inserted. We refer to this occurrence as an "IBLT overpopulation" as it contains too many entries to be useful. A lookup operation is always performed when the cache is accessed to determine the state of the requested block. If this operation fails it is due to previous hash collisions in every hash function in the IBLT; in other words, every cell selected by the hash functions contains more than one entry. In this instance it is impossible to determine whether the block is currently resident in the cache by inspecting the IBLT. It is theoretically possible to test if the block in question has been inserted by using the data recovery procedure outlined above, though this procedure is infeasible to perform in hardware and as such the cache must be flushed to ensure that no duplicate cache entries will be inserted.

4.4 Optimisations

A number of optimisations were made to the basic design of the IBLT cache in order to improve performance. The IBLT, as originally presented, was designed to be used as a software data structure and some modifications were made to reduce the latency of accessing the structure in a hardware environment. The lookup and insertion functions return different values based on the outcome of the operation, which assists in the decision-making process moving forward. The limitation on the amount of data allowed within IBLT cells is also an optimisation as it reduces the amount of failed lookup operations and provides some shortcuts within the logic of the insertion operation. The size of the fields in the IBLT cells have also been modified from the original research to keep the size of the IBLT structure to a minimum.

4.4.1 IBLT Lookup Operation Optimisation

Lookup operations can return one of three results: succeeded, failed, and failed, not inserted. The failed, not inserted case was added to the existing implementation as outlined in the original research to cover the instances where it can be immediately determined that the queried value has not been inserted into the IBLT. An immediate determination can be made when a cell is found with zero entries, or a single entry which does not match the incoming one. This can prevent cache flushes as the subsequent insertion logic no longer needs to ensure that duplicate entries are being added. The succeeded and failed cases are as in the original research, where the requested data is found and not found, respectively.

4.4.2 IBLT Insertion Operation Optimisation

Insertion operations also have some optimisation regarding return values. The original insertion function was defined to always succeed, which creates a number of problems. With the limitations placed on the number of key-value pairs allowed per cell, insertions can now fail. There are two separate failure cases for insertion operations: total failure and failure where only a single eviction is required for a successful insertion.

Total failure occurs when one or more of the selected cells do not have space for an additional key-value pair. The insertion function removes entries from the cache block array in least-recently-used order and reattempts insertion until it succeeds in this case. The other failure case, in which a single entry can be removed for guaranteed insertion, occurs only in a specialised fast-path for the insertion logic when the number of entries per cell is limited to one. Victim entries can be much more easily selected because there is only a single entry allowed per cell. This failure case only occurs when there is one conflicting cell and all others are empty. The conflicting cell can easily be identified as a side-effect of checking the conditions for insertion and subsequently removed. This approach is more efficient than removing cache blocks in least recently

used order, which does not guarantee that a conflicting entry can be inserted without multiple evictions.

4.4.3 IBLT Cell Field Size Reduction

The IBLT cell structure has three fields: key hash, value hash, and count. The original research defined each of these fields to be the size of a memory word, generally 64 bits. This represents a fairly significant amount of wasted storage space, as the values contained in them can be expressed using far fewer bits. This wasted space is particularly evident when the IBLT may contain tens of thousands of cells. Consequently, the size of the fields was restricted to a minimal number of bits.

The key hash field was practically restricted to a maximum of 48 bits by the hash functions, though the actual configurations used in the simulations have fewer. Modern processors tend to use less than 48 bits for tag fields in set-associative caches [textbook, B13, B39], which is what the field is effectively replicating, and therefore assumed to be a reasonable upper bound. It is important that this field is not overly limited because hash collisions will occur more frequently the fewer bits there are to compare.

The value hash field is limited to the number of bits required to contain the maximum index in the cache block array. There is no need for additional bits in this field because the contained value will never be larger than the maximum index of the array. The size of the field is calculated using the following equation:

$$\textit{value hash field bit count} = \lceil \log_2 (\textit{cache block array size} - 1) \rceil$$

The count field can be reduced to only a few bits. Because the number of entries allowed per cell is generally quite small, not many bits are required to keep count. A maximum of three entries, for example, only requires two bits. The following equation is used to calculate the number of bits in the count field:

$$\textit{count field bit count} = \lceil \log_2 (\textit{maximum entries allowed per cell}) \rceil$$

4.5 Parameters

The IBLT cache has a number of parameters which can affect performance. The most important of these is the proportion of the total cache size allocated between the IBLT and the backing array. A larger IBLT will allow for a greater number of mappings to be accurately maintained, though at the expense of storage for the cache blocks and vice versa. An optimal split must be found between these two structures. When comparing an IBLT cache to a classic cache it is important to ensure that the total size of the IBLT cache matches that of the classic cache, as the IBLT will take up space which must be accounted for.

Other parameters have an effect on performance as well. The number of hash functions in the IBLT, for instance, can have an adverse effect if it is too large. Forcing a large number of hash functions will increase the number of partitions in the table, potentially to leading to an increased number of hash collisions if the partitions are too small.

Modifying the number of bits in the IBLT cell fields can save space, though at the cost of some reduction in the accuracy of the mappings. These parameters generally have a smaller effect on the performance of the cache than the total size and the split between the IBLT and the backing array.

Chapter 5

Cache Simulator

5.1 Overview

The cache simulator was written in C++ and contains approximately 3,300 lines of code. It reads the memory traces produced by the capture workflow and simulates the effect of the captured instructions on the cache under test. Various performance metrics are tracked during the simulation, most importantly cache miss rates and miss types.

The simulator supports three cache architectures: fully associative, set associative, and invertible Bloom lookup table caches have been implemented. The fully associative and set associative caches were implemented so data could be generated for comparison against the IBLT cache. Set-associative caches represent the cutting edge of modern cache architecture, providing a useful baseline for performance comparisons. Fully associative caches are ultimately what the IBLT cache is attempting to emulate, so an implementation was created to test the effectiveness of the IBLT design.

The caches can be further combined into a hierarchical structure which is entirely configurable by the user, though data was not generated using hierarchical cache simulations due to project constraints. Each cache includes various parameters which can be modified to affect performance characteristics. Due to the large number of available parameters the simulator accepts specially formatted text files as input in which cache configurations can be defined.

The simulator implementation was verified using synthetic memory traces. These were generated using a custom tool written for this purpose. Each of the three cache structures were determined to behave correctly by examining the resulting miss rates, as they could be tightly controlled in this instance.

The simulator takes as input a cache configuration file and a memory trace file. Output is in the form of .csv files, though statistics data can be written to the console if desired. An accompanying Python script was written to marshal the input and output data for the simulator. The script handles recombining the raw simulation data with the interval weights generated by SimPoint. This task is not handled directly by the simulator

simply because it was less time-consuming to write Python code to find the correct weights file and perform the calculations than it would be to write the equivalent in C++, keeping the code complexity of the simulator to a minimum. The recombination process is discussed fully in Section 6.4.

5.2 Custom Data Structures

A custom doubly-linked list implementation was used in numerous places in the simulator. The maintenance of least-recently-used (LRU) lists was discovered to be a common task which needed to be implemented in several areas of the code. The caches themselves evict blocks based on LRU order and some kinds of statistics tracking also require data to be sorted in this way. The need for an efficient LRU list with directly accessible nodes, similar to a standard array, became apparent.

The solution involves two data structures; a hash map and a linked list. Identifiers (e.g. cache block addresses) for the data are inserted into the hash map as keys and pointers to the linked list nodes are inserted as values. The data is kept in LRU order by removing the corresponding linked list node and moving it to the head of the list when updates occur. Direct accessibility of data is maintained through the hash map as nodes can be directly found through the association it maintains, eliminating the usual need to traverse the linked list until the correct node is found.

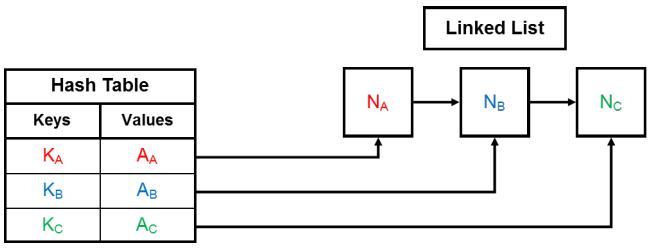

Figure 5.1: Efficient LRU List Diagram

A diagram of the efficient LRU list can be seen in Figure 5.1. Several pieces of data have already been inserted into the data structure. Each key (denoted by $K_{A,B,C}$) is mapped to an accompanying value (denoted by $A_{A,B,C}$) which is an address of a linked list node (denoted by $N_{A,B,C}$). Linked list nodes can be retrieved through the hash table by dereferencing the pointers stored in the value field. The pointers between the linked list nodes can be arbitrarily rearranged without affecting the mappings in the hash table.

The C++ standard library linked list implementations were found to be unsuitable because they do not allow direct access to their nodes. The custom implementation allows pointers and references to nodes so they can be directly accessed. Nodes are not relocated in memory by the list after they are allocated as this would lead to dangling pointers in client code. Arbitrary insertions or removals from the list can be performed while only having a pointer or reference to a single node because it is doubly linked.

The accompanying hash map is an `std::unordered_map` from the C++11 standard library which has average O(1) insertion and retrieval time complexities [7]. An associative data structure which has low access overhead was required to maintain the mapping between keys and linked list nodes and keep lookup time to a minimum. In theory any associative structure which operates on key-value pairs could be used, depending on requirements. For example, the IBLT cache implementation uses this data structure to keep track of cache blocks, except using an IBLT instead of `std::unordered_map`.

This implementation of a directly-accessible, efficient LRU list has proven to work quite well. It is undoubtedly slower to access than a standard array, as it requires a hash map lookup and a pointer dereference to access an element, though it is much more efficient to maintain resident data in a particular order. An early implementation using a standard array required sorting the data at every update, resulting in copying most of the array during the sort as elements were constantly being moved to the head of the list. This implementation manages to maintain the best of both worlds: it has relatively quick lookups and the vast majority of data stays static when nodes are reordered, only a few pointers must be changed.

5.3 Set- and Fully-Associative Cache Implementations

The set-associative cache implementation is of the standard design outlined in Section 2.2.1. Cache block size, the number of sets, and the number of ways are all configurable parameters. Fully associative caches are not implemented separately, but as special cases of set-associative caches where there is a single set and the number of ways is equal to the number of cache blocks. The block replacement policy used in this implementation is least-recently-used. The cache write policies are implemented as write-back, write-allocate policy, the reasoning for which is discussed in Section 2.4.

The implementation for this cache uses a list of cache set objects and distributes access operations between them based on the address calculation as described in Section 2.2.1. Cache sets are configured by the parent cache object with the correct block size and level of associativity when the cache is created. Each cache set maintains a list of the cache blocks which it contains in LRU order. This is achieved through use of the data structure outlined in Section 5.2. Cache sets notify the parent cache object when they evict blocks so the eviction can be properly handled by the rest of the cache hierarchy, if necessary.

5.3.1 Statistics Tracking

This cache tracks statistics for cold misses, capacity misses, conflict misses, read miss rate, and write miss rate. All statistics are accumulated per the intervals defined in the data produced by the memory capture process.

Cold misses are tracked by storing the addresses of all cache blocks which have been

used during the simulation interval and checking new accesses against this list. If a block is not found in the cache or the list previously accessed blocks, then the access is categorised as a cold miss. If the block has been seen before but misses in the cache, then it is determined to be one of the other miss types. Previously unseen blocks are added to the list so future accesses will be correctly categorised. The data structure used to hold the list of cache blocks is an `std::unordered_set` from the C++11 standard library which offers average O(1) time complexity for insertion and retrieval of elements [7]. This is important for performance as the structure is accessed for every simulated instruction.

Capacity misses are tracked by the use of a "moving window" which contains the last n cache blocks used in the sequence of memory accesses, where n is the total number of cache blocks that can be contained in the cache. If a cache miss occurs and the corresponding block is not seen in the window, then the miss is a capacity miss. The number of blocks required to complete the computation at hand has exceeded the available space in the cache, resulting in the missed access. The moving window is implemented with a hash map and a linked list as described in Section 5.2; block addresses are used as keys in the hash map. Entries are moved to the head of the linked list whenever they are accessed, ensuring that the list is always sorted most-recently-used to least-recently used. When the window contains n elements and a new entry must be inserted, the oldest entry at the tail of the linked list is removed and the corresponding key-value pair is deleted from the hash map. The new entry is inserted into the hash map and becomes the new head of the linked list.

Conflict misses are the easiest to classify. If a cache block must be evicted from a set to bring in a new block, the operation results in a conflict miss. Conflict misses are determined as a side effect of the cache block update logic.

Read and write misses are also easily generated as a side effect of the update logic. If any operation misses in the cache, it is categorised into a read or write miss based on whether the operation was a read or a write.

5.3.2 IBLT Cache Implementation

The invertible Bloom lookup table cache is an implementation of the structure described in Section 2.6. It has a number of user-configurable parameters: cache block size, cache block storage array size, the number of IBLT cells, the number of IBLT hash functions, maximum entries allowed per IBLT cell, and the input range of the IBLT hash functions. Statistics on read miss rate, write miss rate, cold miss rate, capacity miss rate, conflict miss rate, the average number of hash functions accessed during a retrieval operation, cache flushes, and the number of IBLT insertion, retrieval, and deletion operations are collected. Like the set-associative cache, the block replacement policy is least-recently-used. The cache uses write-allocate and write-back policies. Other policies were not implemented because it had become apparent that the project would not extend to cover other policies by the time the IBLT cache code was written.

The overall structure of this cache is somewhat similar to the set-associative implementation in that it has a parent cache object which sends requests to a child data storage structure, in this case an IBLT. The IBLT is similarly configured by the parent cache object when the cache is created. The hash functions used by the IBLT are generated externally so new implementations can be easily swapped in and out. All IBLT code which interfaces with the hash functions operates on a set of generic interfaces for this reason.

5.3.3 Hash Functions

The hash functions used in the simulations are generated from the H_1 family of hash functions, as described by Carter and Wegman [2]. This method of hashing was chosen primarily because it guarantees that for every hash function input, the resulting output does not collide with the rest of the output set more than $\frac{|A|}{|B|}$ times on average, where A is the set of inputs and B is the set of outputs [2]. This property ensures hash collisions are kept to a reasonable number, as it could not reasonably be expected for any hash function to experience fewer collisions, given that the set of inputs is larger than the set of outputs [2]. An ideal hash function would experience exactly $\frac{|A|}{|B|}$ collisions if evaluated for the entire input set [2]. Given the exceedingly large number of inputs which are involved during the simulations, it was assumed that the number of hash collisions seen in the simulations would be a reasonable approximation of the predicted average, and thus also the number seen in the ideal case.

The hash functions in the H_1 family are of the form:

$$h_{m,\,n} = ((mx+n) \bmod \mathrm{p}) \bmod b$$

where m and n are integers modulo p and m does not equal zero, p is a prime number greater than or equal to the maximum element in the set of inputs, and b is the size of the output set [2].

A number of hash functions, as defined by the corresponding input parameter, are generated from H_1 when the cache is constructed. Parameters b and p are fixed; new hash functions are created by selecting new values for m and n. Parameter b is the number of cells in one partition of the table in the IBLT, as the hash output must point to a specific cell in the table. Parameter p is the value $1,125,899,839,733,759$, which is the 12th Carol prime, chosen because it is greater than $2^{48}-1$, the maximum value supported by the simulator for hash inputs. Hash inputs cannot take up the full range of a 64-bit integer because the prime used in the hash function must be larger than the maximum input [2]. Clamping the input field as much as possible is desirable because smaller fields save space in the IBLT cells. Full 64-bit ranges are clamped by using modular division, though bit shifting would also be effective and more efficient in practice.

5.3.4 Statistics Tracking

Cold, capacity, read, and write misses are tracked using the same methods as for the set-associative cache; only brief descriptions are given here, see Section 5.3.1 for more detail.

Cold misses are tracked in the same way as in the set-associative cache. A list of all cache blocks seen in the data stream is maintained, the list is checked for each simulated memory access to determine whether the corresponding block has been previously seen. Unseen blocks are classed as cold misses and added to the list for future reference. The list is implemented with an `std::unordered_set` as with the set-associative cache.

Capacity misses are again tracked using the "moving window" approach. Blocks in the window are added or updated as memory accesses occur. The blocks in the window are maintained in LRU order so evictions of the oldest can occur when too many blocks are added to the window. If a memory access causes a block eviction in the window, it is classed as a capacity miss. The moving window is implemented with the same hash map-linked list data structure as in the set-associative cache.

Read and write misses are tracked as they occur during simulated memory operations. If a read or write operation fails, then the corresponding miss is logged in the statistics.

Conflict misses are tracked somewhat differently in the IBLT cache due to the nature of its operation. It is impossible for cache blocks to directly conflict as they would in an associative cache, instead conflicts arise from hash collisions in the IBLT. As such, conflict misses are logged in the statistics when an insertion operation fails and the cache cannot immediately insert a new key-value pair into the IBLT. It should be noted that even if an insertion requires a cache flush or multiple removals of cache blocks, only one conflict miss is logged. Other statistics, notably cache flush counts and IBLT operation counts, are captured separately from conflict miss data as they reflect the behaviour of the cache more accurately than the single metric.

All other statistics are captured through the use of counters. The IBLT itself tracks insertion, lookup, and deletion operations by maintaining separate counters which are incremented when the corresponding operation is executed. The average number of hash functions executed during a lookup operation is calculated similarly, a counter which holds the total number of hash functions executed during retrieval operations is kept updated by the IBLT and is used to divide the total number of such operations to find the average at the end of the simulation run. Cache flushes are tracked by the IBLT cache object through a counter which is incremented when a cache flush occurs.

Chapter 6

Methodology

6.1 Introduction

The process used to collect data in this project required multiple steps and a variety of tools to complete. Figure 6.1 shows this process broken down into the component elements. Each of the sections from Section 6.2 to 6.6 explain the corresponding step in the process.

Figure 6.1: Data Collection Workflow

The first four steps in the process are concerned with capturing memory traces, which are used as inputs for the simulation, the final step. The performance data which is analysed in Section 7 is generated by the simulator.

Several tools were created for use in data collection, all of which were written specifically for the project, with the notable exception of SimPoint. The entire memory capture process is automated by a Python script. The executable to be profiled and any relevant arguments are provided to the script which then executes the process outlined above. The script performs some processing of input and output files between stages, as well as handling the naming of the files produced by the programs in the workflow.

6.1.1 Dynamic Binary Instrumentation

Dynamic binary instrumentation was used extensively during this project due to the need to analyse the behaviour of a running program when profiling memory accesses. Highly detailed analysis of the program's characteristics can be performed with dynamic instrumentation, without the need for disassembly or source code inspection.

Dynamic events which would otherwise be difficult or impossible to study, such as the number of times a basic block is executed, can also be recorded.

A library called Pin, provided by Intel, offers powerful dynamic instrumentation capabilities and was used for the instrumentation done in this project. A tool which uses Pin can register callback functions for a variety of events, with granularity down the execution of a single instruction. Tools which target Pin are frequently referred to as "Pintools" in the Intel documentation, naming which has been adopted for this report.

The Pin library effectively wraps a specialised virtual machine in an interface which provides communication between the Pintool and the hosted program. Instrumentation code is automatically generated and inserted into the instruction stream of the hosted program by Pin based on the commands sent by the Pintool. This approach allows for very efficient profiling, further enabled by Pin's built-in just-in-time compiler, which can recompile frequently executed sections of code, similar to the approach used in the Java virtual machine. The recompilation of code is beneficial due to the fact that the hosted program's code has changed due to extra instructions inserted by Pin, meaning that previously-optimised sequences may no longer be so and can potentially be re-optimised.

6.2 Benchmarks

The SPEC2006 benchmark suite was used as the set of sample programs from which to capture memory traces. SPEC2006 is a benchmark set commonly used in academia and removes the guesswork from picking a representative set of programs with which to conduct experiments. Table 6.1 includes a list of the benchmarks used in the project. Each benchmark is listed with a brief description of its purpose and which arguments were used during the memory capture process, as many have multiple sample inputs to choose from. The arguments used here were chosen because they are each the first in the list of potential inputs for their respective benchmarks, as produced by the SPEC2006 tools. Arguments which have multiple parts are enclosed with double quotes.

The benchmarks listed in Table 6.1 make up the entire integer suite included in SPEC2006. Integer benchmarks were chosen over floating-point because integer instructions make up the majority of baseline processing done in modern computers and it was desirable to test the cache's performance under a slightly more standard workload.

6.3 Basic Block Capture

The first tool used in the memory capture workflow is designed to capture basic block data from the target program. This tool uses Pin to profile the basic blocks; it registers a callback so as to be notified whenever a basic block is executed. As each block is executed, the starting memory address of the block is recorded so the tool can identify

Table 6.1: Benchmarks selected from SPEC2006

Benchmark	Description	Input Arguments
400.perlbench	Perl language interpreter	“checkspam.pl 2500 5 25 11 150 1 1 1 1”
401.bzip2	Data compression	input.combined
403.gcc	C compiler	166.s
429.mcf	Vehicle scheduling	inp.in
445.gobmk	AI which plays Go	13x13.tst
458.hmmer	Database search	ref.txt
462.libquantum	Quantum computer simulation	“1397 8”
464.h264ref	Video compression	“-d foreman_ref_encoder_baseline.cfg”
471.omnetpp	Simulation of a large network	“-i omnetpp.ini”
473.astar	A* pathfinding algorithm	BigLakes2048.cfg
483.xalancbmk	XML processing	“-v t5.xml xalanc.xsl”

blocks if they are seen again in the sequence. The number of instructions in the block is also saved so weighting can be applied to the execution count for the block before it is written to the output. Each block is also given an ordinal number based on the first time it was seen in the sequence of blocks, to be used as an identifier later in the memory capture process.

The execution of the target program is split into intervals, measured by the count of completed instructions. This is the format of the data expected by SimPoint, which is the next tool used in the process. Intervals are explained in-depth in Section 6.4.1.

As the tool runs, it accumulates an execution count for each basic block it finds. After the requisite number of instructions for each interval has been reached the data is saved and all counters are reset for the next interval. The output of this program is a text file containing a list of basic block ordinals and a weighted execution count for each block seen in the interval, for all intervals in the program. The weighted count is calculated by multiplying the number of times a block was executed by the number of instructions in the block.

6.4 SimPoint

6.4.1 Background

SimPoint is a methodology used to isolate intervals of operation in a given program or system which are representative of a larger sample of execution [5]. Using SimPoint to perform this analysis greatly reduces the amount of trace data that needs to be executed in the simulator. This was deemed necessary due to the extreme size (in the order of terabytes) of the raw memory traces recorded during the preliminary phase of the project. It would have been infeasible to run these complete traces in the simulator as the execution time required would have been impractical.

The authors of the SimPoint methodology provide a sample implementation free for use with academic and non-profit projects. This sample implementation (version 3.2) was used in this project, though some minor modification was required to support recent GCC versions.

SimPoint works by partitioning the execution of a program into a series of intervals and using a clustering algorithm to identify intervals which are similar [5]. From each cluster the most representative interval is chosen and written to the output with an accompanying weighting factor. The weighting factor is used to properly recombine data which has been generated using the SimPoint output [5].

Intervals can be of fixed or variable length depending on the requirements of the project [5]. Variable length intervals are used to reduce the amount of computation required to identify the most representative intervals, though potentially with the expense of less accurate output [5]. Interval "length" is measured in the number of instructions executed during the interval [5].

An interval in SimPoint is represented by a frequency vector [5]. The frequency vectors used in this project measure the execution of basic blocks [5]. The values contained in the frequency vector are execution counts for each basic block in the input data, weighted by the number of instructions in the block [5]. Some of these values may be zero as not every basic block is guaranteed to be executed during each interval [5]. A basic block frequency vector is thus representative of an interval of execution as it contains the number of instructions executed, broken down into approximately equal sections of code. Frequency vectors are compared using Euclidean distance, treating each vector as a point in n-dimensional space, where n is the number of dimensions in each vector [5].

The clustering algorithm used in SimPoint is an implementation of k-means [5]. The frequency vectors are pre-processed before clustering; each vector is normalised (so the sum of all elements is equal to one) and the dimension is reduced [5]. Dimensionality reduction is done by creating a projection matrix with random values and fewer dimensions, then using it to project the frequency vectors into a new space [5]. The resulting reduction in size produces vectors which are quicker to process and reduces the difficulty encountered when attempting to classify sets of data with a large number of dimensions [5]. After the frequency vectors are pre-processed, the k-means algorithm is used to cluster the data [5]. Various values are tried for k and the best clustering is chosen based on the Bayesian Information Criterion [5]. BIC determines how well the clustering approximates the dataset [5]. The clustering with the best BIC score is chosen as the final clustering [5]. From each cluster in the final set a representative interval is chosen [5]. The representative interval is selected by determining the centroid of the cluster and picking the interval closest to it [5]. A weighting value, which represents the proportion of execution the selected interval represents, is also calculated [5]. Finally, the selected intervals and their respective weightings are written to the output [5].

6.4.2 Usage

SimPoint is used to analyse the basic block trace produced by step 2 in the memory capture process. A list of representative intervals of execution and matching weighting factors are generated by SimPoint, as described above. SimPoint produces a list of intervals to be profiled, along with matching weighting factors. These intervals are pre-processed by the Python script before being sent to the next stage in the process. A separate input file is produced with the weighting factors removed and the intervals sorted in ascending order for ease of use by the next tool.

Fixed length intervals were used in this project as the computation time used by SimPoint was not considered to be an issue. Computation done by SimPoint was informally observed to take no more than several minutes on a consumer laptop. An interval length of 10,000,000 was used, which is the default value recommended by the authors of SimPoint. Usage of this interval length produced final memory traces of approximately 1-2GB in size containing 200 to 400 million memory operations.

6.5 Memory Trace Capture

The final step to produce memory traces is accomplished by feeding the SimPoint output into a second Pintool which actually records the memory instructions themselves. This Pintool works similarly to the first except that it operates at an instruction-level granularity as well as basic block-level. The output of this tool is a stream of memory accesses captured from the target program, divided into the intervals defined by the first tool in the process.

This Pintool registers callbacks for both basic blocks and instruction executions. Basic block instrumentation is used to correlate the SimPoint output with the execution of the program. To do this the tool must keep track of how many instructions have been executed by the target program because the input intervals are measured in numbers of instructions. It is more efficient to count the number of instructions executed at the basic block-level rather than at the instruction-level because the instrumentation callback does not need to be invoked for every instruction, only for each basic block. Thus, instructions are counted at a basic block level of granularity.

The tool's main purpose is instrumenting instructions which access memory that fall into the intervals defined by the SimPoint output. Instrumentation is toggled on and off depending if the current instruction count falls within an interval which has been identified as one to record data for. Each instruction that is executed by the program when instrumentation is turned on is checked for memory operands. Only instructions which operate on memory are instrumented as other instruction types are irrelevant to cache performance. The checking of individual operands is required due to x86 architecture quirks where certain instructions may have multiple loads or stores to memory that must be accounted for.

This tool finishes by emitting a trace file with the instructions that it has recorded, divided into intervals. The total number of instructions per interval will be less than

the size used at the start of the memory capture process (e.g. 10,000,000) because not every instruction will have accessed memory.

6.6 Simulations

The final step in the process is to run simulations. The simulator takes the trace files generated by the memory capture process and a cache configuration file as arguments. Cache configurations are reflective of the selected parameters in Section 7.2. Due to their small size, configurations for set associative and full associative caches were written by hand, though configurations for the IBLT cache were generated automatically using a Python script.

Approximately 30,000 simulations were run to generate the data for this project. The simulations were run in parallel on the "Eddie3" compute cluster provided by the Edinburgh Compute and Data Facility.

Chapter 7

Evaluation

7.1 Design Space

The total design space for the caches is quite large. The cache types are discussed separately because they all have different parameters and design space sizes. The allowable values listed below are listed in their entirety for completeness, in many cases values are only limited by the integer sizes used by the simulator. It should be noted that some combinations of parameters are disallowed due to the limitations of set-associative indexing. The listed ranges are inclusive.

Table 7.1: Fully Associative Cache Parameters

Parameter	Allowed Parameters
Cache block size (in bytes)	$1, \ldots, 2^{31} - 1$
Cache block count	$1, \ldots, 2^{31} - 1$

Table 7.2: Set Associative Cache Parameters

Parameter	Allowed Parameters
Cache block size (in bytes)	$1, \ldots, 2^{31} - 1$
Cache set count	$1, \ldots, 2^{31} - 1$
Cache set associativity	$1, \ldots, 2^{31} - 1$

Table 7.3: IBLT Cache Parameters

Parameter	Allowed Parameters
Cache block size (in bytes)	$1, \ldots, 2^{31} - 1$
IBLT hash function count	$1, \ldots, 2^{31} - 1$
Cache block count	$1, \ldots, 2^{31} - 1$
Size of the IBLT max. hashes per cell field (in bits)	$1, \ldots, 64$
Size of the IBLT hash function input range (in bits)	$1, \ldots, 48$

7.2 Samples

Given the extreme size of the parameter space, random sampling was ruled out, even taking into account unreasonable values (e.g. a cache with $2^{31} - 1$ way associativity). Ranges of acceptable values were selected based on typical hardware cache configurations.

The sizes for each cache vary from 4KB to 1024KB, with a sample at every power of 2. These sizes were chosen because they are representative of L1 cache sizes which might commonly be found in today's devices. A 4KB cache might be found in a low-power embedded device while a 1024KB cache might be seen in a high-performance processor for a desktop or server. Configurations for IBLT caches were generated while keeping the total cache size static, meaning that the actual amount of data which can be stored in the cache is less because a portion of the cache is taken up by the space required by the IBLT. This was done so configurations could be compared in the context of similar size.

A cache block size of 64 bytes was used for all caches. This is the standard cache block size commonly found in processors and textbook descriptions and it was deemed unnecessary to simulate others as there are a significant number of parameters already in effect.

A cache configuration was generated for every combination in the tables below.

Table 7.4: Fully Associative Cache Selected Parameters

Parameter	Selected Values
Cache block size (in bytes)	64
Cache block count	$64,\ 128, \ldots,\ 16384$

The cache set parameters in Table 7.4 were generated by dividing each cache size by every associativity parameter.

Table 7.5: Set Associative Cache Selected Parameters

Parameter	Selected Values
Cache block size (in bytes)	64
Cache set associativity	$2,\ 4,\ 8$
Cache sets	$\left\{ \frac{x}{y} \middle\vert x \in 64,\ 128,\ \ldots,\ 16384; y \in 2,\ 4,\ 8 \right\}$

Cache configurations for the IBLT cache were generated in two rounds, the second after the initial data was analysed for gaps in the space of selected parameters.

The total cache sizes and the block size are kept the same as the other cache types. IBLT hash functions were stepped by 2 from a starting value of 2 to a maximum of 8. These parameters were found to adequately capture the behaviour of hash functions in the cache so no more values were added.

Table 7.6: IBLT Cache Selected Parameters

Parameter	Selected Values
Total cache size (in KB)	4, 8, 16, 32, 64, 128, 256, 512, 1024
Cache block size (in bytes)	64
IBLT hash function count : IBLT partition size	2 : { 1, ..., 16 } 4 : { 1, ..., 8 } 6 : { 1, ..., 6 } 8 : { 1, ..., 4 }
Size of the IBLT max. hashes per cell field (in bits)	1, 2, 3, 4
Size of the IBLT hash function input range (in bits)	16, 32

The partition size values are multiplied by the number of cache blocks in the backing array to get the number of cells in a partition. Partition size parameters were originally set to 1-4 for all hash function count parameters. A gap the parameter space was identified where the IBLT partition sizes were not being adequately explored in the initial configurations. To rectify this, more variables were added for the partition size. It was noted in the initial round of analysis that performance in the IBLT cache degraded when the ratio of IBLT cells to cache blocks was more than 32. New configurations were made inside this bound, hence the somewhat strange distribution of hash function counts and partition sizes.

The maximum entries per cell values range from 1-4 (in bits). This allows for maximums of 1, 3, 7, and 15 entries in each IBLT cell. These were not modified after the initial round of samples as adding increasing numbers of entries allowed per IBLT cell is detrimental to cache performance.

The hash function input ranges were initially set to be 16 or 32 bits. Input ranges were shown to have a minimal effect on cache performance during the initial round of simulations so no more parameters were added.

7.3 Results

7.3.1 Invertible Bloom Lookup Table Cache Characterisation

The simulation data specific to the invertible Bloom lookup table cache is analysed in this section. We attempt to ascertain the most effective IBLT cache configuration and analyse the performance of the cache in general.

In general, the IBLT cache shares performance trends with current cache designs. The larger the cache is, the lower the corresponding miss rate.

Figure 7.1 shows this in more detail. It should be noted that all IBLT cache configurations are captured within the graph, so some variation is to be expected in the results. The data in this graph is an average of the total miss rates of identical IBLT cache configurations, broken down by cache size. The plotted miss rates therefore take all

benchmarks into account, as each cache configuration was simulated once for each benchmark.

Multiple statistics are captured within the graph; the boxes represent the middle 50% of data, red lines indicate the median, and black horizontal bars at the end of the vertical lines represent edges of the 25th and 75th percentiles of data. The repeated blue pluses are outliers; data points which lie outside the 25th and 75th percentiles. The same formatting is used for all boxplots in the report.

From inspecting the graph, it can be seen that the data reflects the intuitive reasoning in which a larger cache provides better performance. As cache sizes increase, there is a downward trend in miss rates, bottoming out at approximately 10%. It should also be noted that there are fairly significant diminishing returns present in cache size increases. This also intuitively makes sense, even a theoretical infinite cache would still have some missed accesses (cold misses), so the miss rate will slowly approach zero the larger the cache becomes.

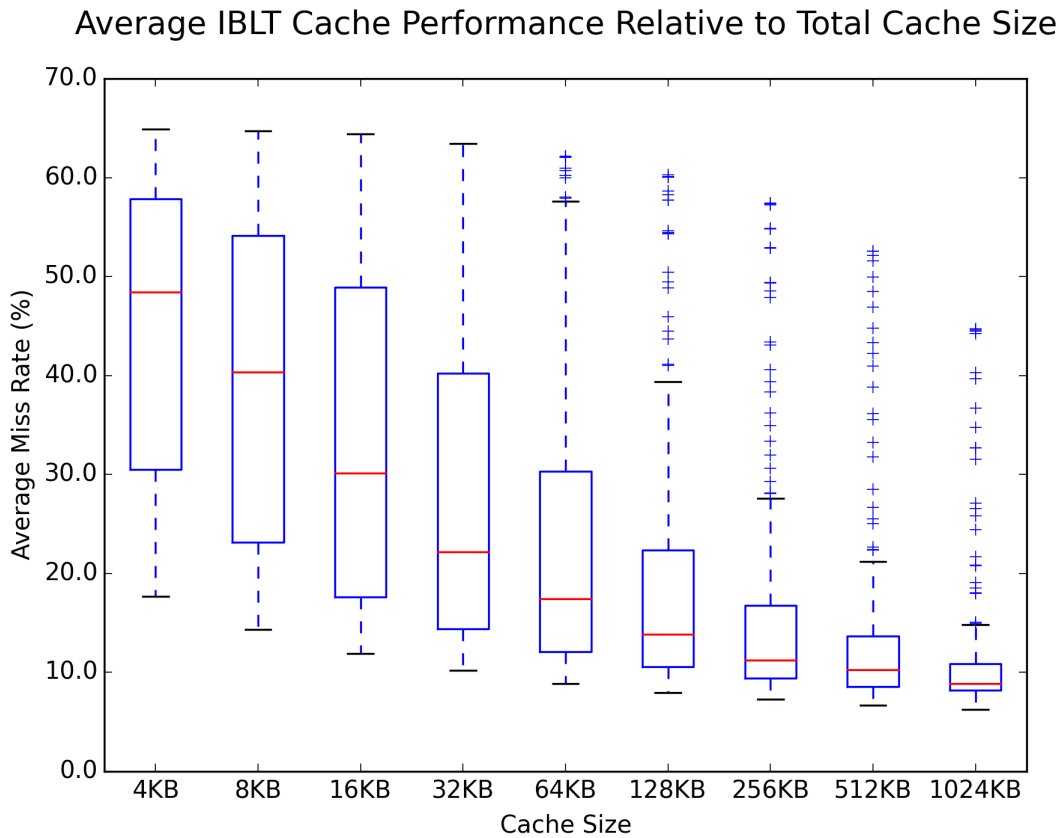

Figure 7.1: IBLT Cache Performance Relative to Total Cache Size

Figure 7.2 shows the average miss rate of the IBLT cache for each benchmark. All IBLT samples for each benchmark were averaged to produce the results plotted in the graph, broken down by total cache size. From this data we can gain a general understanding of the cache utilisation of each benchmark, which is useful for interpreting some of the following graphs in this section.

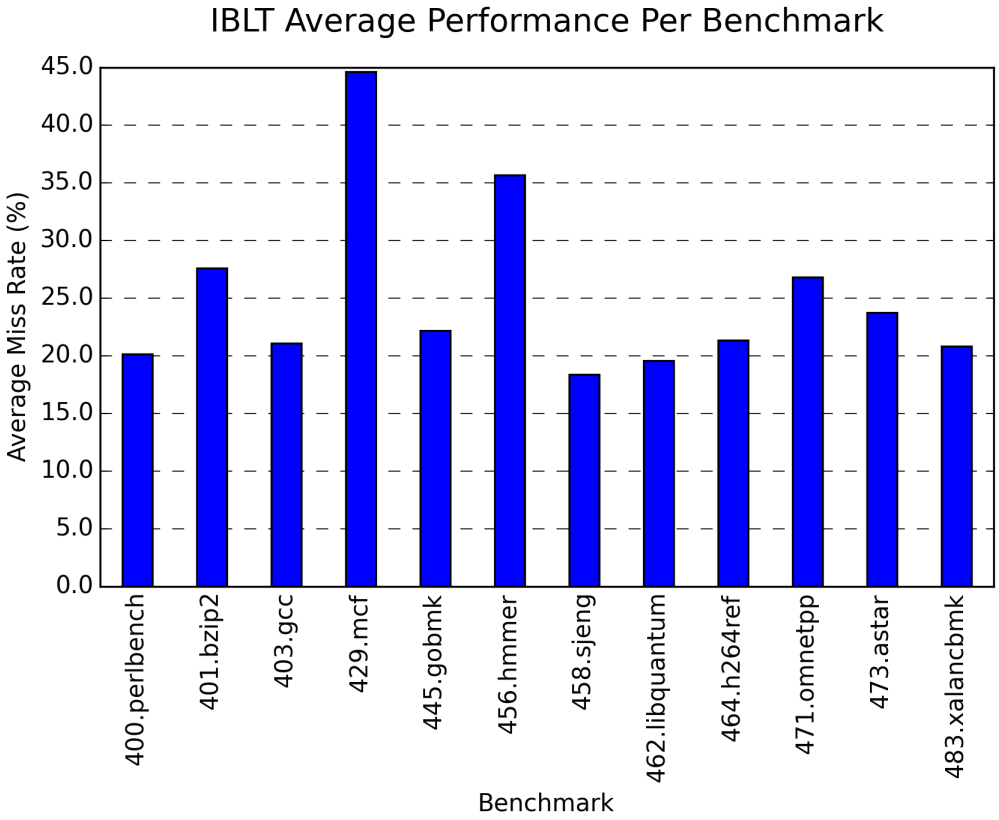

Figure 7.2: IBLT Average Performance Per Benchmark

7.3.1.1 Best Case Performance

Figure 7.3 shows the average performance of the best-performing cache for each size. The best-performing cache was found by taking the configuration with the lowest average miss rate across all benchmarks, for each cache size.

This graph shows that the cache size is a fairly strong indicator of overall performance. The same trend of diminishing returns is present; a doubling of the cache size produces an approximately 10%-15% reduction in overall miss rate over the previous size.

7.3.1.2 Mapping Storage vs. Data Storage

The data in Figure 7.4 show the relationship between the size of the backing array in the IBLT cache to the total size of the cache. Because the IBLT itself takes up fairly significant space, some must be taken from the cache block storage to fit in the allocated space. This graph is broken down by cache size and is aggregated over all IBLT cache configurations and benchmarks.

The trend lines show a consistent relationship between performance and the proportion of the cache allocated to the backing array. Caches of all sizes experience a significant increase in performance as the backing array increases in size. Larger caches appear

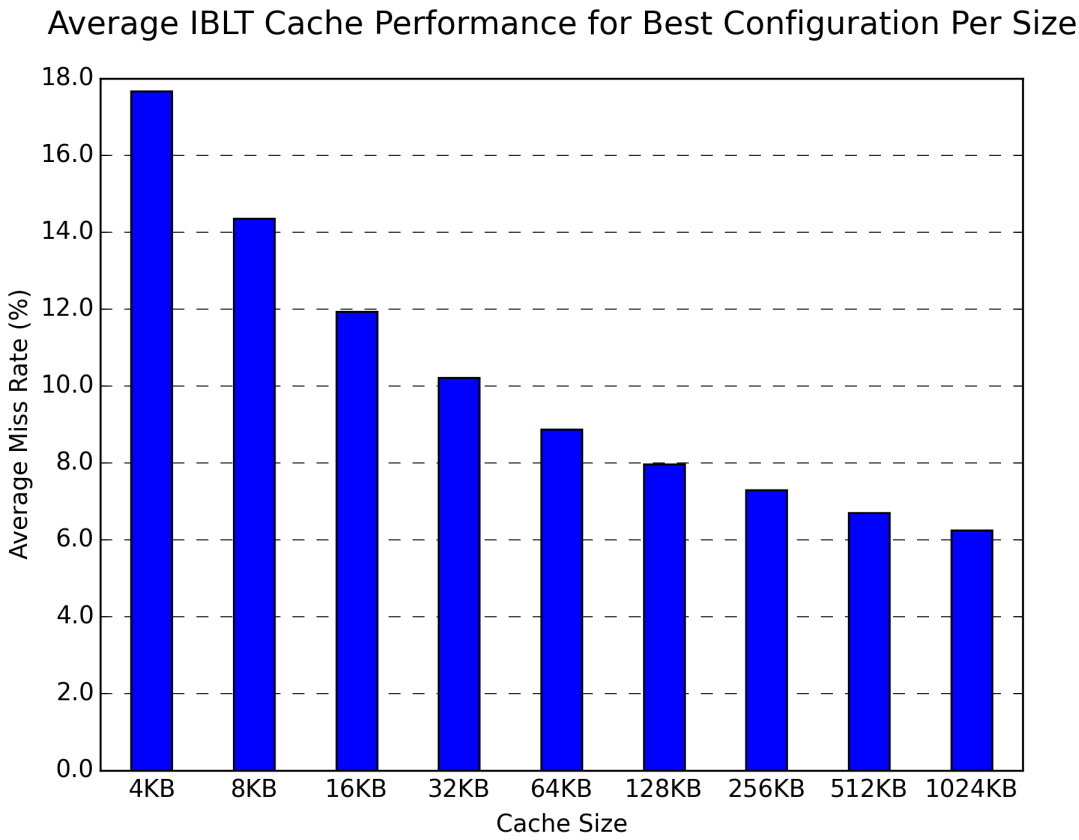

Figure 7.3: Average IBLT Cache Performance for Best Configuration Per Size

to be more stable in terms of this allocation. The largest cache (1024KB), for instance, reaches its best level of performance at an approximate 50% split between IBLT size and backing array size, and maintains this level of performance until the backing array is increased beyond approximately 85% of the cache. Smaller caches, on the other hand, do not show the same resilience. The smallest of the caches shown (4-8KB) have a very small window of optimal performance.

These trends seem to follow what could be expected of the IBLT cache. Because the IBLT must share a portion of the total size, performance suffers when it is made too large because there are not enough blocks in the cache to hold the data. Conversely, allocating too much space to the backing array takes away space from the IBLT which it needs to operate. If the IBLT is too small, then fewer mappings can accurately maintained and block evictions and cache flushes become much more likely. These effects can be seen more strongly in the smaller caches simply because there is less space to work with; sensitivity to bad configurations is increased as a consequence. There is an obvious trend between the size of a cache and its ability to tolerate varying allocations of IBLT and backing array size.

From Figure 7.4 we can determine that there is a balance between the cache space

[1]The trend lines used here were modelled using 4th degree polynomials with an average R^2 value of 0.301. Increasing the degree of the polynomial beyond 4 did change the R^2 value significantly (the approximate change was 0.01); a 4th degree polynomial was chosen for this reason.

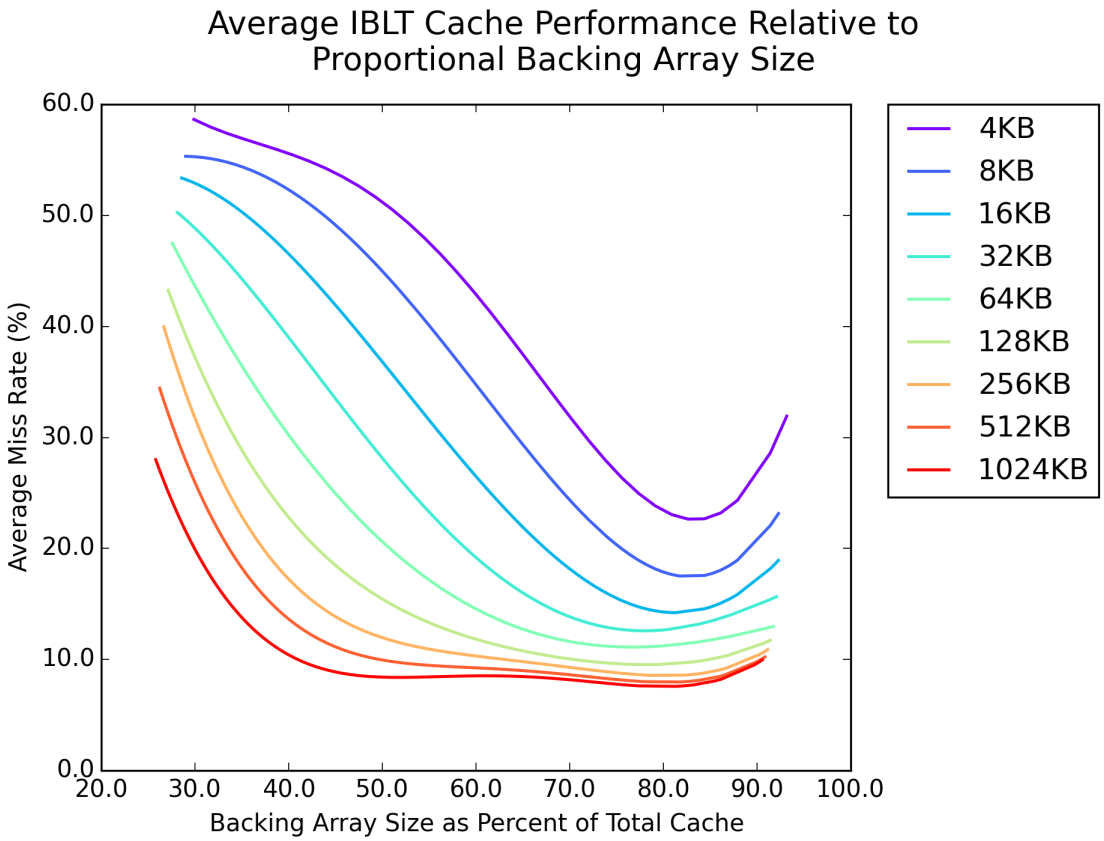

Figure 7.4: IBLT Cache Performance Relative to Proportional Backing Array Size[1]

allocated to the backing array and to the IBLT. If one or the other has too much space, cache performance suffers because the reduced structure will not have the requisite area to operate effectively.

The relationship between the size of the IBLT as a proportion of the total and miss rate is not shown because it is the mirror image of the representation shown in Figure 7.4.

7.3.1.3 IBLT Hash Function Effectiveness

Figure 7.5 shows the relationship between the number of hash functions in an IBLT and cache performance. The data in this graph are averages of the total miss rates of identical cache configurations across all benchmarks, divided by total cache size and shown relative to the number of hash functions used by the IBLT.

There is a slight increase in the miss rate of the cache when the number of hash functions is increased, particularly in the smaller caches. This is most likely due to the fact that a larger number of hash functions requires additional partitioning of the IBLT. The increase in partitions means that the partitions must become smaller, requiring that the hash functions decrease the size of their output spaces. This decrease in output space size will lead to more hash collisions in the IBLT, reducing overall performance of the cache.

Larger caches are much less affected by the number of hash functions. Almost no change in performance is seen between the differences in hash function counts. Very slight reductions in miss rate are apparent in the largest two caches, indicating that perhaps small gains can be made from increasing the number of hash functions if there is sufficient space to support them.

In general, it seems that increasing the number of hash functions does not have much effect on the performance of the cache. It can be seen in Section 7.3.1.2 that there is a much stronger correlation between cache size and performance than the number of hash functions.

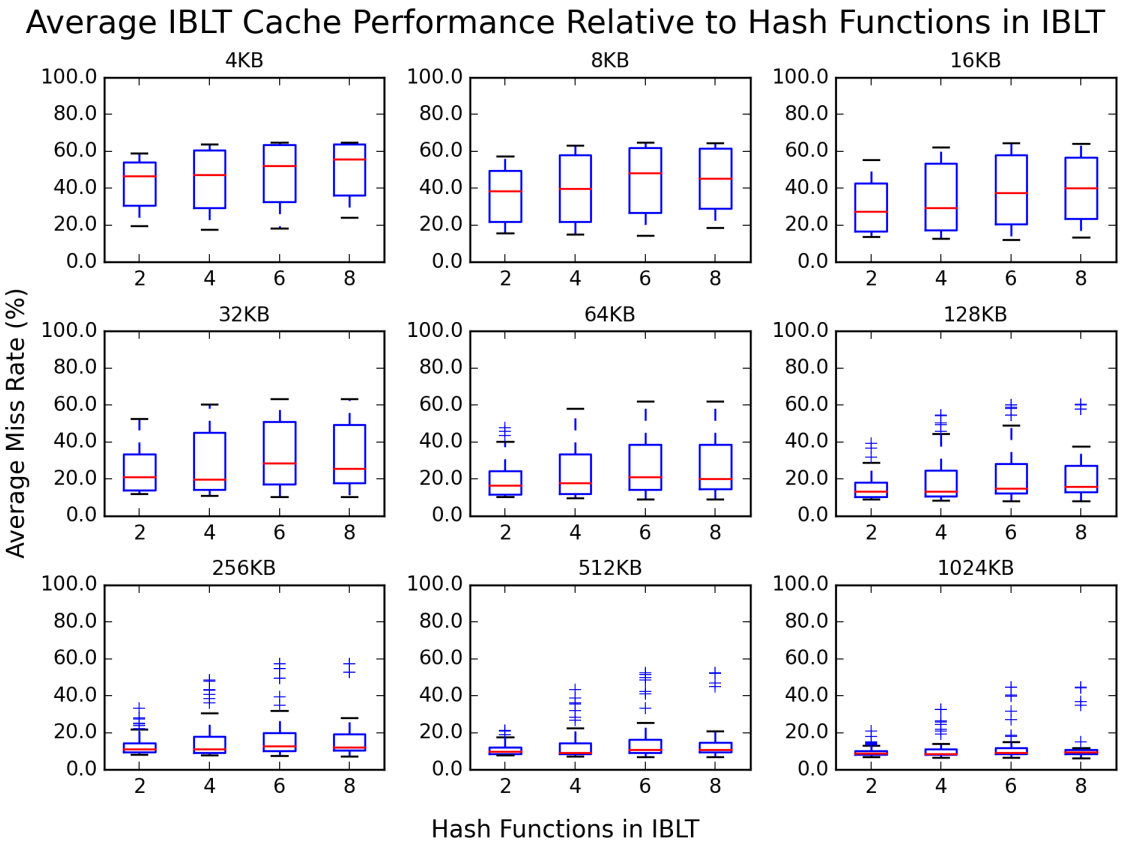

Figure 7.5: IBLT Cache Performance Relative to Hash Functions in IBLT

7.3.1.4 Effects of Hash Function Input Range

The data in Figure 7.6 represent the relationship between the number of bits used in the IBLT hash function inputs and the performance of the cache. As with the previous graph, the data shown are averages of identical cache configurations across benchmarks, grouped by cache size and relative to the input range of the hash functions used by the IBLT.

The trends shown in Figure 7.6 are broadly similar to those seen in Figure 7.5. There is a slight increase in miss rate in the smaller caches and almost no change in the larger

caches when the size of the input space is increased. This can be explained by the fact that the increase in input range requires significantly more bits to store the key hash in the IBLT cell, for apparently limited returns. The increase in table size overshadows any additional hashing accuracy gained from the increase in input space.

Unfortunately, it seems that there is not much difference in cache performance when the input range is varied, meaning as a parameter it is somewhat unimportant. A finely tuned parameter might provide a small performance increase depending on the exact configuration but it generally seems to have little effect on the cache as a whole.

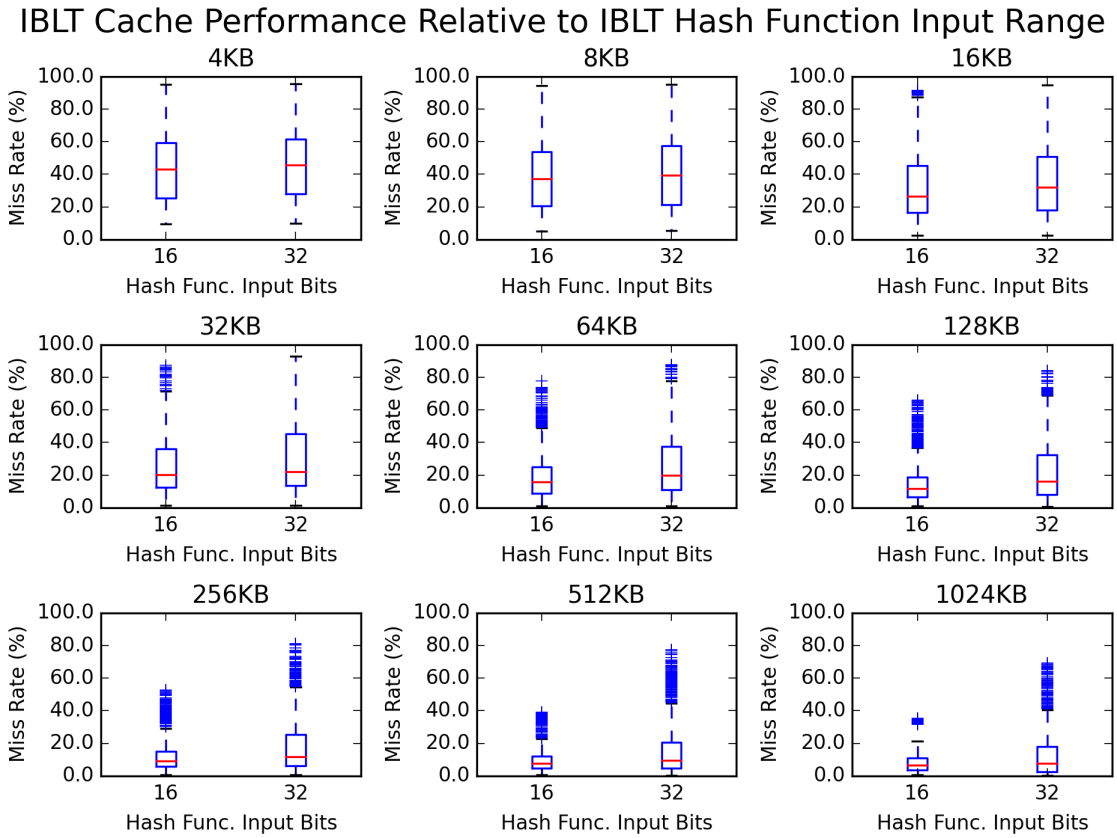

Figure 7.6: IBLT Cache Performance Relative to IBLT Hash Function Input Range

7.3.1.5 Maximum Entries Allowed per IBLT Cell

Figure 7.7 displays the relationship between the number of key-value pairs allowed per cell in the IBLT and cache miss rate. As with the previous graph, the data shown are averages of identical cache configuration miss rates across benchmarks, divided by total cache size and relative to the total number of entries allowed in each IBLT cell.

The trends seem to follow what can be generally expected based in the IBLT design. The more entries allowed per cell, the higher chance there is for colliding keys to be stored in the same cell, potentially irrecoverably. Given the difficulty that the IBLT

cache has in resolving hash collisions between keys, as it cannot copy the IBLT for key recovery, limiting the number of entries per cell seems to be a fairly reliable stopgap.

The usual trend of a larger cache providing better performance is also present here. The performance is also much more stable in the larger caches, allowing for additional entries per cell has little to no effect on the miss rate of the cache. This is most likely due to the fact that the larger caches can more effectively distribute IBLT entries, leaving the number allowed per cell mostly irrelevant.

The smaller caches show a distinct decrease in performance when the number of entries per cell is increased beyond one. This implies that limiting the number of insertions and subsequently evicting more entries is more effective than allowing multiple insertions per cell and attempting to resolve conflicts later, in the case where size is at a premium.

From this graph we can ascertain that a small IBLT cache should have a maximum of one or two entries per cell, where for larger caches this parameter makes little difference. It would be reasonable to limit the larger caches as well, as they use the same internal logic as the smaller caches. Larger sizes seem to paper over the difficulty in handling multiple entries per cell rather than resolving the problem.

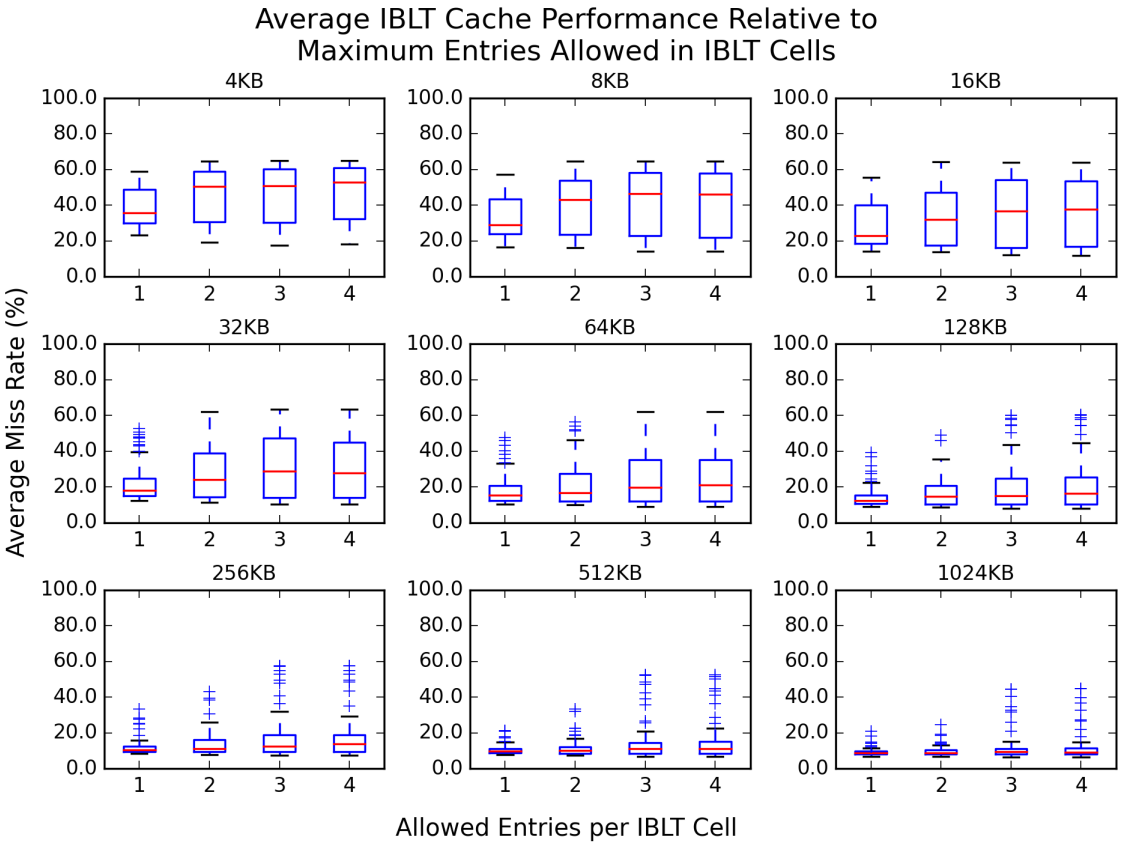

Figure 7.7: IBLT Cache Performance Relative to Maximum Entries in IBLT Cells

7.3.1.6 IBLT Overpopulations

The data presented in Figure 7.8 show the number of cache flushes per memory access caused by the IBLT (IBLT overpopulations) relative to its proportional size in the total cache. The trend lines in this graph were generated by aggregating all samples, split by total cache size

The IBLT must be flushed in certain instances when it becomes overpopulated; when it is impossible to recover a desired key-value pair contained within it due to hash collisions between keys. These flushes can be mitigated by increasing the size of the IBLT so fewer collisions occur.

The graph shows a distinct reduction in overpopulations as the IBLT size is increased. Total cache size has less of an effect than in some of the other data presented, though the smallest cache is still heavily influenced by the proportional size of the IBLT. In the larger caches there are effectively no cache flushes, indicating that overpopulations can be mostly solved by increasing the cache size. It should be noted that some overpopulations would need to be tolerated for most cache sizes when using the approximate 70%-30% split between backing array and IBLT which was seen to give best performance in Section 7.3.1.2.

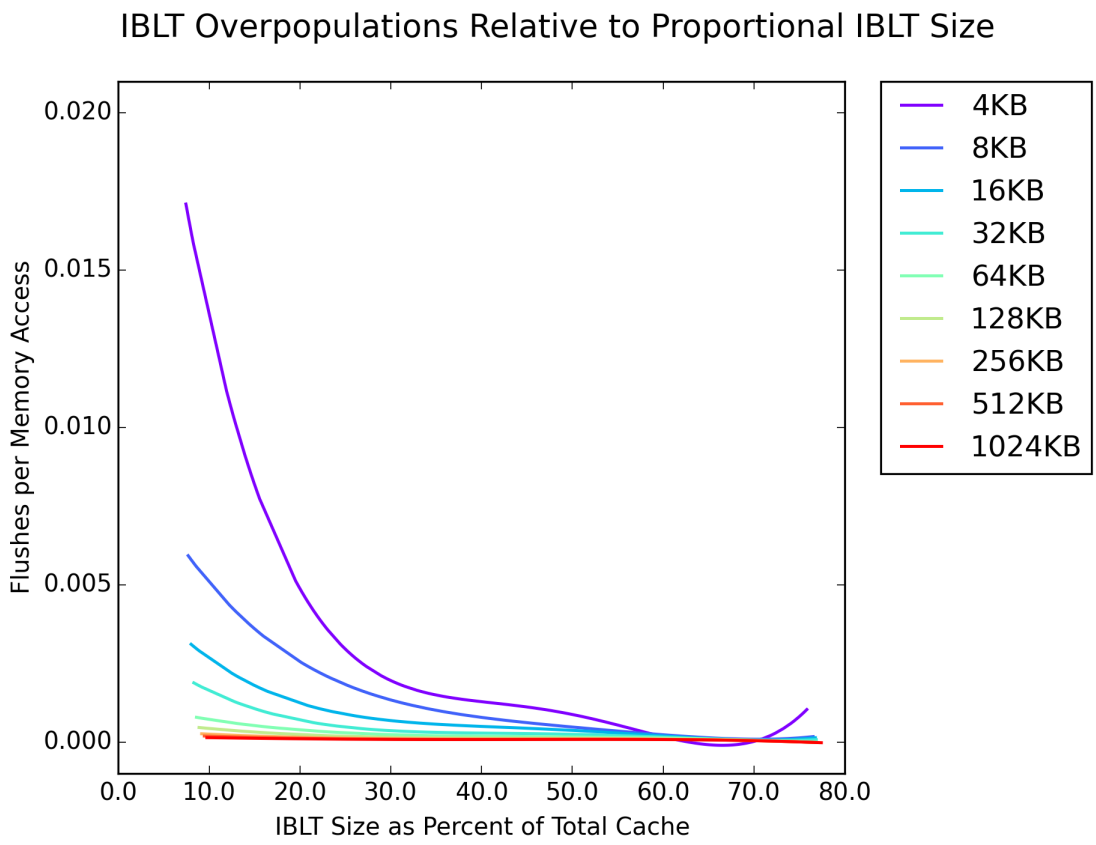

Figure 7.8: IBLT Overpopulations Relative to Proportional IBLT Size[2]

[2]The trend lines in this graph are 4th degree polynomials with an average R^2 value of 0.215, using

7.3.1.7 Average Hash Function Accesses During IBLT Lookup Operations

The data displayed in Figure 7.9 show the relationship between the average number of IBLT hash functions accessed during a lookup operation and the proportion of the total cache occupied by the IBLT. The data is an aggregation of the average number of hash functions used during a lookup operation (a statistic provided directly by the simulator), split by total cache size.

When the IBLT performs a lookup, it will return early if it finds the requested data before it has executed all of the hash functions. As such, efficiencies can be gained when the IBLT uses fewer hash functions to find data.

The larger caches have better overall performance, though all caches show a distinct decrease in performance when the IBLT size takes up more than approximately 60% of the cache. This appears to be a turning point where cache misses have increased to a level such that all of the hash functions must be evaluated for a lookup because the block cannot be found.

From this graph, we can ascertain that the latency of cache lookups in larger caches is less than that of smaller caches because of the reduced number of hash functions which are on average required for to perform a lookup. Hash functions could be run in parallel, though there are concerns with the additional energy which would be consumed by always executing every hash function. There is a definite trade-off between cache latency and energy consumed.

Overall, there is an optimal level of hash functions accessed when the IBLT is around 30% of the cache. Most of the trend lines have a downturn at this point, or at least a reduction in increase. Thirty percent is also the ideal IBLT size as seen in the previous graphs, making this point particularly important.

7.3.2 Comparison of Classic and IBLT Caches

Figure 7.10 shows the performance of the IBLT cache versus set and fully associative caches. All cache configurations are represented in this graph. The data shown in the graph is an averaged total miss rates of identically sized cache configurations broken down by benchmark.

Unfortunately, it is apparent that the IBLT cache performs consistently worse than both set associative caches and fully associative caches. In some instances, it has performance comparable to the other cache types, though this is most likely due to the program not stressing the cache fully. Other benchmarks stress the cache much more significantly and the IBLT cache fails to perform at a satisfactory level.

---

the same reasoning as the above graphs. The dip into negative flushes seen in the trend line for the 4KB cache is due to a slight overfitting of the data, not an actual error.

[3]The trend lines in this graph are 5th degree polynomials, using the same reasoning as the above graphs. The R^2 value is 0.191.

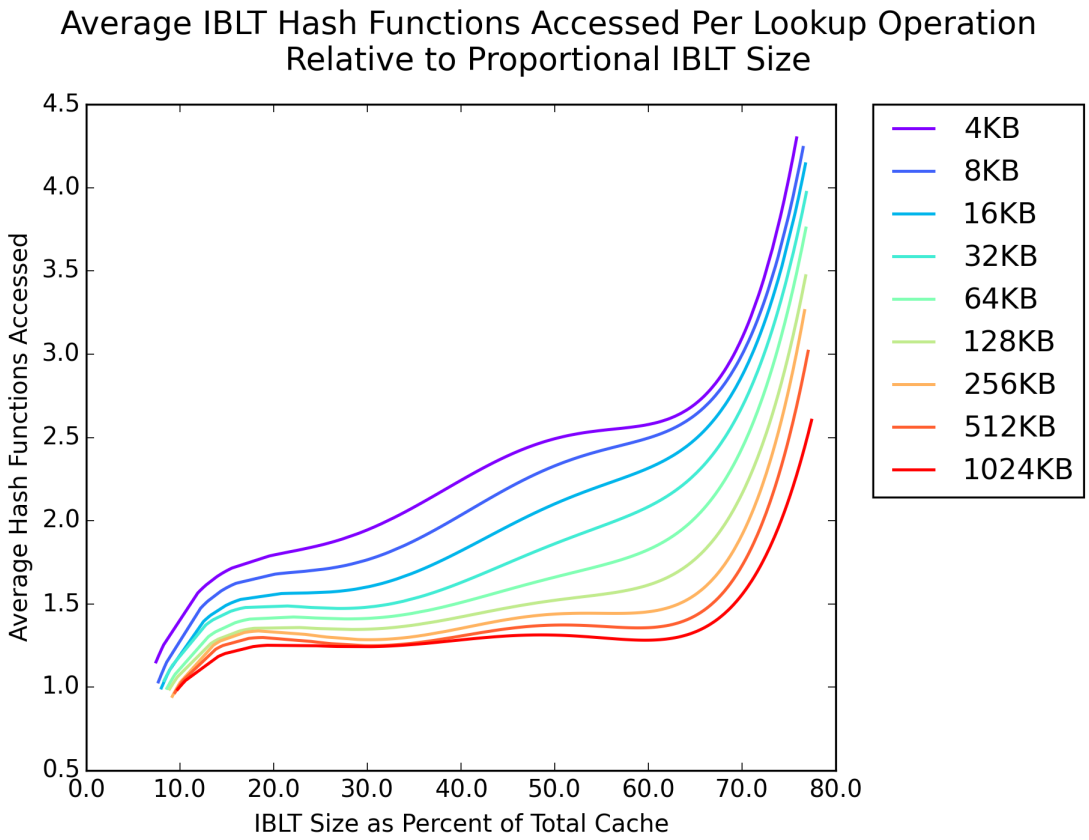

Figure 7.9: Average IBLT Hash Functions Accessed Per Lookup Operation Relative to Proportional IBLT Size[3]

Figure 7.11 shows a slightly simplified view of Figure 7.10. The data in the graph shows the lowest average total miss rate over all benchmarks, broken down by cache size. In all cases, the IBLT cache is outperformed by the other two caches.

Figure 7.12 shows averaged cache misses broken down by the classification of the miss. The data shown are averages over all benchmarks of the miss breakdown statistics of identical cache configurations, split by total cache size.

The data shown in Figure 7.12 concern the three miss types: cold, capacity, and conflict misses, as discussed in Section 2.3. As expected, there are no conflict misses in the fully associative caches. The goal of reducing the number of conflicts in the IBLT cache has been achieved, as capacity misses are more heavily weighted than conflict misses. Unfortunately, the total number of misses caused by the cache is higher, as indicated by the shorter cold miss bars (cold misses are unavoidable, they should be the greatest proportion of misses).

7.3.3 Summary

From the results of the simulations, it has been shown that the IBLT cache does not perform at a better or equivalent rate than the set or fully associative caches. Var-

ious parameters of the IBLT were explored and it was found that the proportion of the total cache size allocated between the IBLT and the backing array was the most important indicator of cache performance. The optimal split for these parameters is approximately 30%-70%, respectively.

Some statistics concerning IBLT performance were shown, notably the average number of hash functions executed during lookup operations and the number of overpopulations which occur. These data backed up the intuitive reasoning that larger IBLTs perform better than smaller ones.

The size of the fields in the IBLT cells were seen to not be too important for the overall performance of the cache as they were overshadowed by the more influential parameters.

Lastly, it was shown that the proportion of conflict misses was reduced in the IBLT cache over a set-associative cache, though this was generally irrelevant as the overall number of misses was higher.

It is conjectured that the IBLT cache did not perform as desired due to two factors. The IBLT itself takes up space in the cache, limiting the number of blocks which can be stored to less than that of a similarly-sized classic cache. There are also issues with conflicts in the IBLT which sometimes require the cache to be flushed. These two issues can have significant performance impacts on hit rates, particularly in smaller caches where the IBLT cache performs especially badly.

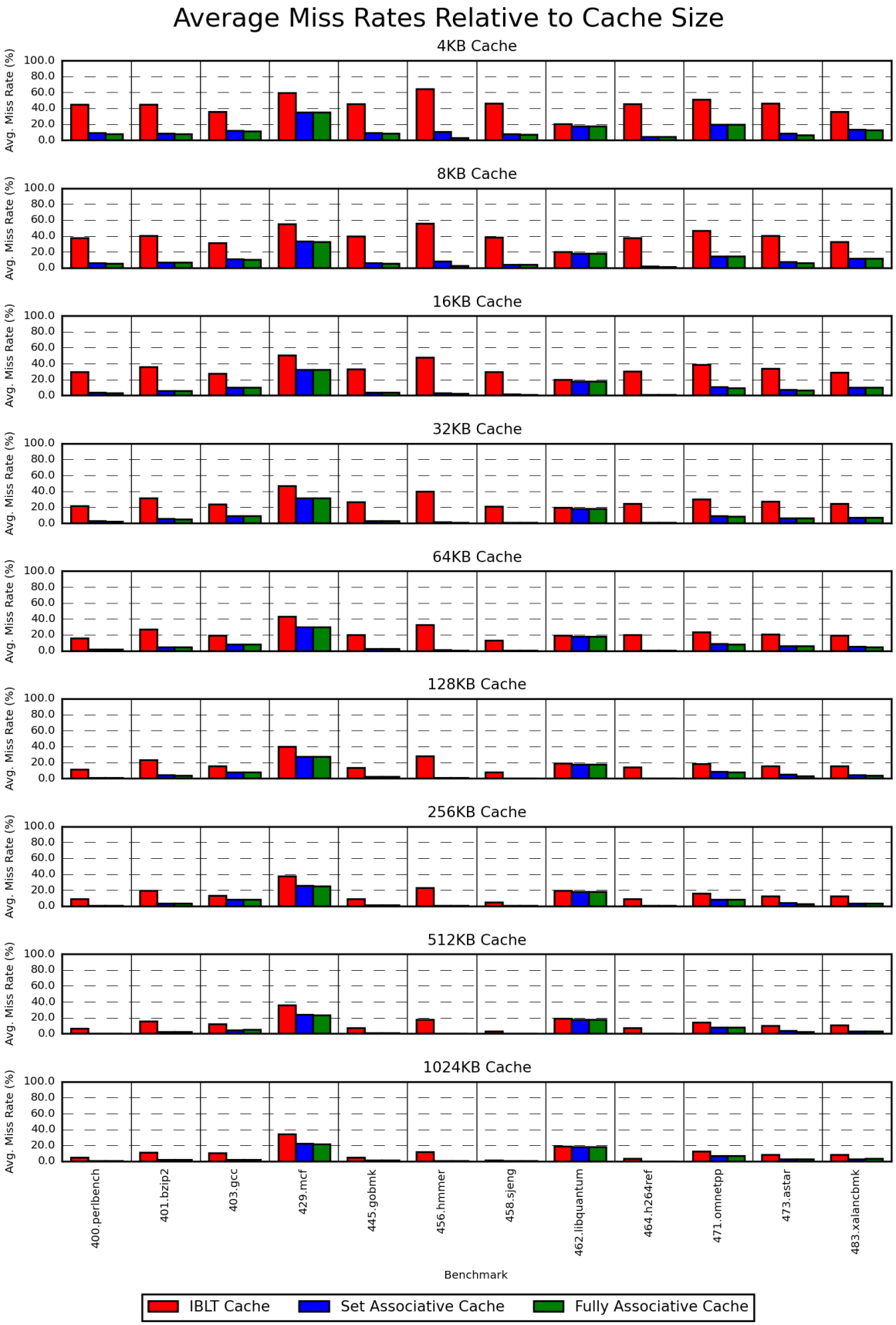

Figure 7.10: Average Miss Rates Relative to Cache Size

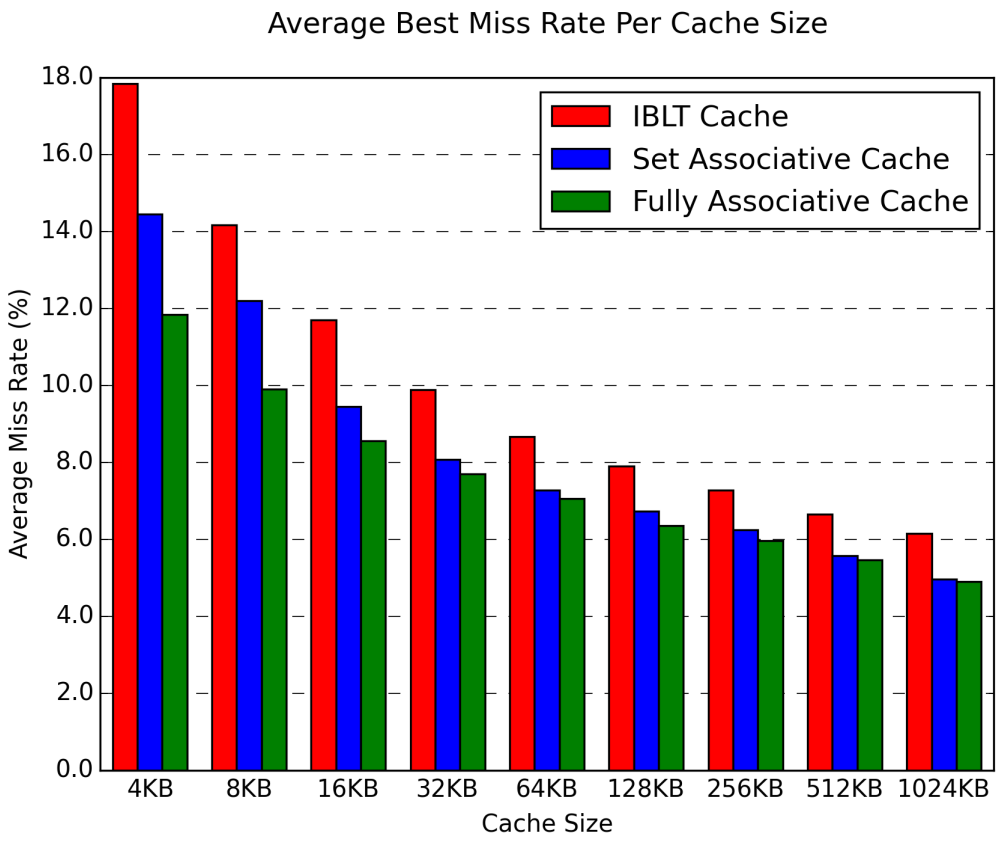

Figure 7.11: Average Best Miss Rate Per Cache Size

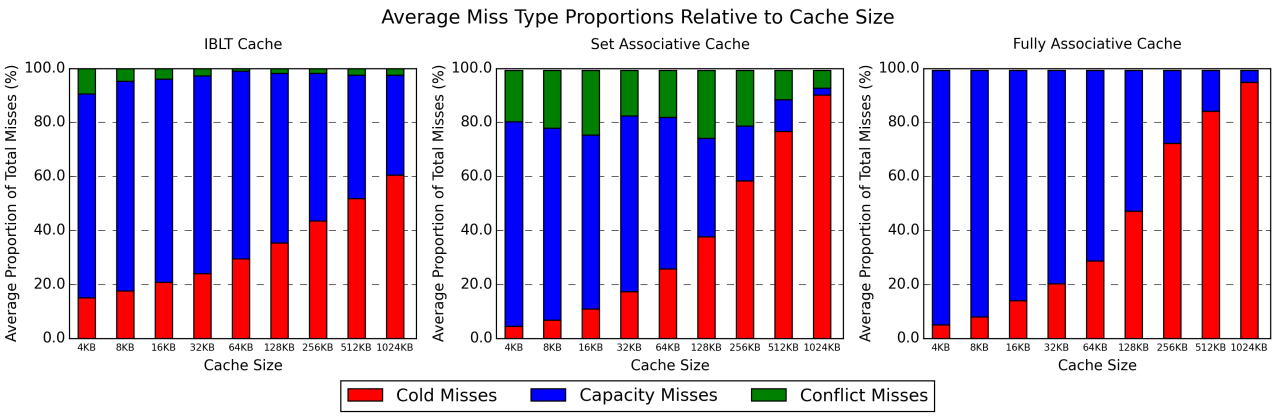

Figure 7.12: Miss Type Proportions Relative to Cache Size

Chapter 8

Conclusion

8.1 Future Work

Due to the less-than-promising results shown in this report, there may be little reason to continue work in the area of IBLT caches. However, there are areas of design unexplored by this project which could potentially produce worthwhile research.

The implementation of the hash functions was not extensively explored in this project. While the H_1 family used in this implementation of the IBLT cache gives promising guarantees about distribution of outputs, no further exploration was done. A project which compares cache performance across multiple hash function implementations may produce some more positive results.

There could also be potential for research in resolving the cache flushes caused by IBLT overpopulations. An effective algorithm for resolving overpopulations beyond simply limiting the number of entries per IBLT cell could potentially allow the cache to perform at a level comparable to a classic cache.

8.2 Critical Analysis

In total, nearly all of the work done during this project was used directly in the final results. The work in the project was undertaken roughly in the order of the methodology outlined in Section 6. This approach reduced the potential for lost work as each part of the project was completed in succession.

Some extra work was done during the implementation of the simulator. The associative caches have support for multiple write and write miss policies as it was assumed that different policies would be explored at this stage in the project. A hierarchical cache framework was also implemented for the same reason. It later became apparent that simply exploring an L1 implementation of the IBLT cache would provide sufficient scope for the research in this project.

8.3 Summary

This report introduced the idea of using an invertible Bloom lookup table as the mapping structure in a CPU cache in hopes of reducing the number of conflicts in cache block mappings over a classic cache, potentially making the new design more efficient. Some aspects of the IBLT were revised to adapt it to a hardware environment, notably limiting the sizes of the data storage fields in the structure and reducing the number of entries allowed in the table. A design for the IBLT cache itself was produced, including specialised logic for inserting and removing blocks in the cache. The logic of these functions as in the original IBLT research required modification for safe use in the cache. Insertion operations were augmented to fail when blocks could not safely be inserted in the cache and it was ensured that deletion operations were only executed when it was safe to do so.

Testing of the IBLT cache design was done through the use of a simulator, written specifically for this project. Memory traces from a subset of benchmarks the SPEC2006 suite were captured through the use of custom tools and used as the stream simulated memory accesses during the simulations. A number of cache configurations were created and performance data was generated from them using the simulator.

Analysis of the simulation data was performed by comparing the performance of the IBLT cache against that of set and fully associative caches. A characterisation of the IBLT cache's behaviour was also performed, analysing the effects of different configuration parameters. It was found that the proportion of the cache allocated between the IBLT and the backing array should be approximately 30%-70%, respectively, while other parameters have relatively little effect on performance. This analysis was done with the aid of graphs, a number of which were generated to illustrate the various performance aspects of the IBLT cache.

Unfortunately, it was found that the IBLT cache does not perform at the same level as associative caches. Miss rates in the IBLT cache are consistently higher in all cases when gauged against a comparable associative cache.

Bibliography

[1] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[2] J.Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143 – 154, 1979.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[4] Michael T. Goodrich and Michael Mitzenmacher. Invertible bloom lookup tables. *CoRR*, abs/1101.2245, 2011.

[5] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. Simpoint 3.0: Faster and more flexible program analysis. In *Journal of Instruction Level Parallelism*, 2005.

[6] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[7] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, 2012.

[8] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, May 2004.

[9] André Seznec. A case for two-way skewed-associative caches. *SIGARCH Comput. Archit. News*, 21(2):169–178, May 1993.