Area-Efficient Near-Associative Memories on FPGAs

Udit Dhawan
udit@seas.upenn.edu

André DeHon
andre@acm.org

Department of Electrical and Systems Engineering
University of Pennsylvania
200 S. 33rd St.
Philadelphia, PA 19104

ABSTRACT

Associative memories can map sparsely used keys to values with low latency but can incur heavy area overheads. The lack of customized hardware for associative memories in today's mainstream FPGAs exacerbates the overhead cost of building these memories using the fixed address match BRAMs. In this paper, we develop a new, FPGA-friendly, memory architecture based on a multiple hash scheme that is able to achieve near-associative performance (less than 5% of evictions due to conflicts) without the area overheads of a fully associative memory on FPGAs. Using the proposed architecture as a 64KB L1 data cache, we show that it is able to achieve near-associative miss-rates while consuming 6-7× less FPGA memory resources for a set of benchmark programs from the SPEC2006 suite than fully associative memories generated by the Xilinx Coregen tool. Benefits increase with match width, allowing area reduction up to 100×. At the same time, the new architecture has lower latency than the fully associative memory—3.7 ns for a 1024-entry flat version or 6.1 ns for an area-efficient version compared to 8.8 ns for a fully associative memory for a 64b key.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*Associative Memories*; B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Gate Arrays*; E.2 [**Data**]: Data Storage Representations—*Hash-table representations*

General Terms

Algorithms, Design, Performance

Keywords

FPGA; BRAM; Associative Memory; CAM; Cache; Hashing

1. INTRODUCTION

With increasing use of high frequency soft processors on FPGAs (*e.g.*, [26, 12]) and an increasing use of FPGAs for processor emulation (*e.g.*, [22, 21, 20, 13]), we need to be able to implement high-performance memory sub-systems on FPGAs (such as caches and TLBs). However, FPGAs are notoriously poor at supporting the associative memories that are often needed in high-performance processors. For example, a recent work [21] observed:

> "Lesson 2: *The major challenges when mapping ASIC-style RTL for a CMP system on an FPGA are highly associative memory structures...*"

The Content-Addressable Memories (CAMs) needed to implement associative memories cannot be built efficiently out of LUTs and the hardwired SRAM blocks provided in modern, mainstream FPGAs (*e.g.* Xilinx BRAM, Altera M4K). While Xilinx Coregen can produce parameterized CAMs [23], they can have enormous overheads. For example, on a recent Xilinx Virtex 6 device with 36Kbit Block RAMs (BRAMs), a 512-entry CAM with a 40-bit key requires 64 BRAMs to perform the match, despite the fact that 512, 64-bit entries can be stored in a single BRAM. That is, *the overhead for implementing the match portion for the fully associative memory on this FPGA is* 64× *the stored memory capacity.* The overheads increase with the match width. [25] shows that fully associative memories implemented on the Stratix architecture have comparably high overheads.

We show how to implement maps with substantially less overhead in comparison to a fully associative memory using BRAMs. We achieve these savings, in part, by implementing memories that are only **statistically** guaranteed to be conflict free. As such, we call them near-associative memories. Specifically, we use a multiple hash scheme [1, 14] based on a generalization of [7] that can be efficiently implemented on top of BRAMs. We further develop efficient replacement policies exploiting the power of choice [1, 14, 16, 11]. This allows us to reduce the conflict miss probability to below 0.03% for the above 512-entry CAM while using only 12 total BRAMs.

Our novel contributions include:

- Customization of the table-based Perfect Hash scheme [7] for efficient implementation on FPGAs (Sec. 3.2)
- FPGA-customized memory architecture that can be tuned to trade-off BRAM usage with conflict miss rate (Sec. 3)
- Analytic derivation of optimal sparsity factor (Sec. 3.7)
- Analytic characterization of capacity (Sec. 3.5) and miss rate (Sec. 3.4), showing that the architecture can achieve very low (≈ 0.05%) conflict miss rates with substantially fewer BRAMs than Xilinx Coregen-style associative memories

FPGA'13, February 11–13, 2013, Monterey, California, USA.
Copyright 2013 ACM 978-1-4503-1887-7/13/02 ...$15.00.

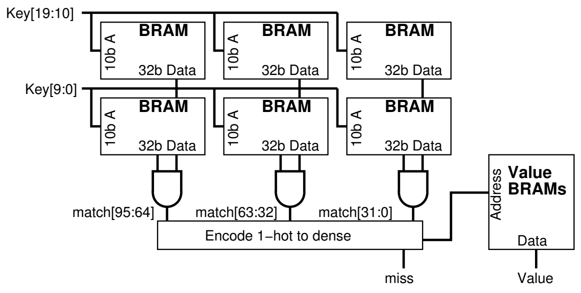

Figure 1: Fully Associative Memory using BRAMs (20b match, 96-entry example shown)

- Identification of a family of replacement policies and characterization of their performance, area, and cycle time implications (Sec. 4)
- Empirical quantitative comparison of the area and performance of our new memory organization against fully associative and set-associative memories (Sec. 5)

2. BACKGROUND

2.1 Associative Memories

An associative memory provides a mapping between a match key and a data value. The set of match keys can be sparse compared to the universe of potential keys. An associative memory of capacity M can hold any M entries; as long as the capacity is not exceeded, there are no conflicts among stored key-value pairs in an associative memory. If the system does need to store a new key-value pair when the memory is at capacity, the memory controller is free to choose any existing key-value pair for replacement, typically based on a policy such as least-recently used (LRU), least-recently inserted (LRI), or least-frequently used (LFU).

However, this freedom comes at a high area and energy cost, since the hardware needs to perform programmable, parallel matches in the entire memory against the incoming key. As a result, fully associative memories are typically only feasible for shallow memories with small keys such as translation look-aside buffers. Nevertheless, the use of fully associative memories or content-addressable memories (CAMs) can be crucial to enhance performance in many applications like network routing [15] and dictionary lookups for pattern matching and data compression/decompression [5].

2.2 Fully Associative Memories on FPGAs

Building fully associative memories or CAMs on modern, mainstream FPGAs is expensive because the memory resources present on these devices do not naturally support the structures needed to implement a fully associative memory. In a custom implementation, a CAM address-match cell is programmable so it can match against any key. However, in an ordinary SRAM array, the address-match cell is fixed. Since FPGAs only contain ordinary SRAM blocks, CAMs must be built out of logic and these embedded SRAMs (*e.g.*, BRAMs), as shown in Fig. 1.

In order to evaluate how area-inefficient building CAMs on FPGAs using SRAM blocks can be, we created custom CAMs the way Xilinx Coregen program suggests [23] for a fully associative memory for a Virtex 6 FPGA (xc6vlx240t-2 device) [24]. This device contains 416, 36Kbit Block RAMs, which can be organized as 18×2048, 36×1024 or 72×512 memories (where there is a parity bit for each byte stored).

In order to build an m-wide, n-deep CAM on a Virtex FPGA, Coregen organizes it as a matrix with 2^m rows (a row each for all the *possible* match keys) and n columns (a column for each of the locations for an associated value). Each matrix cell is a single bit where, for each possible match key, a 1 in a cell means that the data is at the location specified by that column, otherwise, it is not. Using such an organization, one can fit a 10-bit wide, 32-entry deep CAM match unit in a single BRAM (using a 36×1024 configuration) [23]. In order to build deeper CAMs, one can use multiple BRAMs and send in the same 10 bits to be matched to each BRAM. This requires $\lceil \frac{n}{32} \rceil$ BRAMs, where n is the depth of the CAM. Building this further, if the data to be matched is wider than 10 bits, then we can use multiple 10-bit match BRAM sets and build a final AND-tree to see if there was a complete match or not. This means that the total number of BRAMs needed to build the match unit for a m-wide, n-deep CAM using this organization is:

$$\#\ \text{BRAMs} = \left\lceil \frac{m}{10} \right\rceil \times \left\lceil \frac{n}{32} \right\rceil \qquad (1)$$

Now, let us assume that we are building a fully associative memory for storing 64-bit wide data values associated with 64-bit match keys. Table 1 shows the number of BRAMs needed to implement this memory for different depths. We observe that, for a memory deeper than 1024 entries, we run out of the BRAMs available on the device (shown in red italics text). Consequently, we would like to know how to build maps much more compactly than the normal fully associative memory design, especially when the key-width is large or a high capacity is needed.

Table 1: BRAMs consumed for a Fully Associative Memory with 64b match key and 64b data values

Depth	BRAMs consumed for Match Unit	BRAMs consumed for Data	Total BRAMs consumed
256	56	1	57
512	112	1	113
1024	224	2	226
2048	*448*	*4*	*452*

3. A NEAR-ASSOCIATIVE MEMORY

The Coregen-style associative memories are inefficient for three reasons: (1) they demand dense storage of 10b match subfields—which typically means sparse storage of keys since we must allocate space for *potential* keys rather than present keys, (2) they demand sparse (one-hot) encoding of results, (3) they demand re-encoding of the one-hot results into a dense address and indirection to retrieve the actual data value. Ideally, we would like to be able to do almost the opposite: (1) densely store only present key-values pairs, (2) densely store results, (3) directly retrieve the data from a single memory lookup. Taking these as our targets, we create a hash-based memory system with an efficient implementation around BRAMs, called the Dynamic Multi-Hash Cache Architecture or `dMHC`, that can yield near-associative performance.

3.1 Basic Approach

Ideally, we would like to be able to compute a simple function of all the bits of the key, get the address where the data value is stored, and fetch the stored value in a single memory

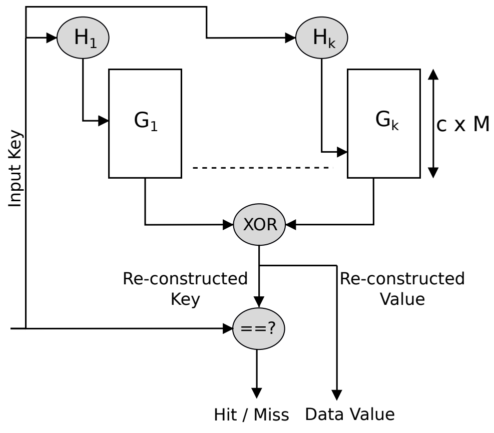

Figure 2: A Generic `dMHC(k,c)`

lookup. A direct-mapped cache works roughly like this, except it can have high conflict rates since many keys will map to a single memory location. Similarly, a typical hash table functions in a similar manner, but stores many data values linked together in the same location; finding the intended value from the slot can sometimes take many memory operations or considerable hardware. If we make the hash table very sparse, we can reduce the probability of conflicts, and hence expected number of key-value pairs mapped in a single hash slot, at the expense of a much larger table.

Instead, we build on an idea that comes from Bloom Filters [3], Multihash Tables [1, 14], and Perfect Hash functions [7]: *use multiple orthogonal hashes.* Bloom Filters determine set membership, with a possibility of having false positives, by hashing the input key with k independent hash functions and setting (reading) a 1-bit memory indexed by each of hash function. On a set membership test (read), the bits are AND'ed together. If any bits are not set, that's a demonstration that the key in question is not in the set. If all the bits are set, either the key is in the set or we have a false-positive due to the fact that multiple keys happened to have set all the hash bits associated with this key.

We define the *sparsity factor*, c, as the ratio of the depth of the memory tables to the number of values stored in the tables. If the hash functions are independent and map all keys to random memory entries, then the probability of a key getting a false hit in any memory is less than $\frac{1}{c}$. The probability of a false hit in **all** k memories is less than c^{-k}, which can be made small by increasing c or k—we'll see how to best do this later in Sec. 3.7.

As originally defined, the Bloom Filter only identifies set membership, but we want to store (and retrieve) a value as well. We can extend the idea by storing the associated data value in the memory along with the single presence bit. Now, AND'ing the presence bits tells us if we have the value. However, we cannot AND the values and get the right result. Instead we will show in the following sections that we can reasonably XOR the values to retrieve the appropriate result. In many applications, we will want to know when a false-positive has occurred. To do that, we will further need to store the key in the memories along with the data value, like we store the address in a direct-mapped memory to know when we actually have a cache hit.

3.2 Hardware Organization

The top-level hardware organization of our `dMHC` architecture is shown in Fig. 2. We use k mutually orthogonal hash functions, H_1 to H_k, and a programmable lookup table called a G table for each hash function. These G tables are used to store the key-value pairs. Each of the G tables is made $c\times$ deeper (*i.e.*, made sparse) than the total capacity (number of entries) in the memory, where c is an integer (a power of 2 in our implementations). In the rest of the paper, we refer to a generic instance of our architecture as `dMHC(k,c)` with k hash functions and a sparsity factor of c.

For an input key D, we divide it into $p = \lceil |D|/n \rceil$ equal parts, n being the number of bits in the final hash value ($n = \log_2(c \times M)$, where M is the total number of entries in the memory). Our family of orthogonal hash functions look like the expression shown in Eq. (2):

$$H_i(D) = \text{XOR}(\phi_{i,1}(D_1), \ldots\ \phi_{i,p}(D_p), P) \qquad (2)$$

Here P is an arbitrary n-bit prime number, and $\phi_i(x)$ is a bit permutation function such as bit-reversal and pairwise bit swap. For H_1, we can use the identity function: $\phi_{1,j}(x) = x$. For H_i, $i > 1$, we might set $\phi_{i,j}(x) = \text{rotate}(x, r(i,j))$, where $r(i,j)$ is different for every value of j within a given i. This family of hash functions was shown to possess the properties of uniform randomness and good local dispersion in [18]. These properties make it highly unlikely for similar keys to have the same hashed values and be stored in the same locations. At the same time, these hash functions allow a simple FPGA implementation (Table 2).

The G tables store the key-value pairs in a distributed form. Each key-value pair is mapped into k G table entries that can later be combined together to form the original key-value pair. We use XOR for this purpose. Traditional hash tables and set-associative caches demand that we compare the input key to the stored keys in each of the k slots (ways) and use the comparison result to select the appropriate entry. By storing the values this way, we reduce the latency to recover the key-value pair. As shown in the Fig. 2, the G table outputs are fed to an XOR-reduce tree to re-construct the key-value pair from the k pieces read off the G tables. In [7], modulo arithmetic is used both for the hash functions and for combining the G table outputs. We replace modulo arithmetic with XORs to make these computations more efficient for LUT-based implementation. The change to XOR's forces us to use power-of-two G tables and M entries.

3.3 Access Operation

Here we explain the operation of our memory for a read access (write access is similar) for a `dMHC(k,c)` instance. The memory receives a read operation along with the key to be looked up. First, we compute k hash values on the key to get h_i (i in 1...k). Each h_i is an index into the G table G_i and from each table we read the key field key_i ($= G_i[h_i].key$) and the value field val_i ($= G_i[h_i].val$) stored at that index. Then, we can re-construct the stored key and the value as:

$$key = \text{XOR}_1^k(key_i) \qquad (3)$$

$$val = \text{XOR}_1^k(val_i) \qquad (4)$$

Next, we compare the re-constructed key against the input key to check if they both match. In case they match, the key-value pair is present in the memory and we can return the data value at the same time; otherwise, the key-value pair is not in the memory and we get a miss. In case of a miss, we then yield to a memory controller to service the miss, which we explain later in the Sec. 4.

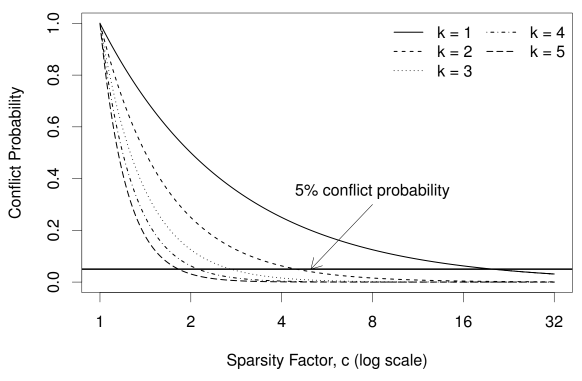

Figure 3: Conflict Probabilities

3.4 Conflict Probability Analysis

Now that we have described the hardware architecture and operation of our dMHC architecture, we present an analytical characterization on a parameterized dMHC(k,c) instance to show how we can reduce the conflict probability to arbitrarily small values.

In the dMHC architecture, there is a conflict when all the G_i table entries that an input key hashes into are in use by one or more key-value pairs already present in the memory. The probability of an input key colliding with the present key-value pairs in a single G table is approximately:

$$P_{collide} < \frac{\text{Capacity}}{\text{G Table Depth}} = \frac{|M|}{|G|} = \frac{1}{c} \qquad (5)$$

Since all the hash functions are mutually orthogonal, the probability that an input key collides in all the hash functions is:

$$P_{k-collide} < (P_{collide})^k \propto \frac{1}{c^k} \qquad (6)$$

This suggests that by choosing high values of parameters k and c, we can make the probability $P_{k-collide}$ arbitrarily small. Consequently, the common case is that new key-value pairs do not have a complete collision and can be inserted easily.

We can further define the conflict miss ratio as:

$$P_{conflict\_miss} = \frac{\text{Conflict Eviction Count}}{\text{Total Misses}} \qquad (7)$$

$P_{conflict\_miss}$ is zero for a fully associative memory. Fig. 3 plots Eq. (6) to show how the conflict miss probability falls as a function of the sparsity factor, c for a particular number of hash functions, k.

3.5 dMHC Area Model

In order to achieve near-associativity, dMHC could require high values of the k and c parameters. In order quantify the FPGA resources consumed by a generic dMHC instance and compare them with those consumed by a fully associative memory, we develop an FPGA area model for a dMHC(k,c) design. In a dMHC design, BRAMs are consumed by the G tables used for storing the different pieces of the key-value pairs (we also need to store the original key-value pairs as we will explain later in Sec. 4, but we skip that for the time being). For simplicity of our area model, we assume that all the BRAMs are used in a 36×1024 configuration, giving us an effective data width of 32 bits per BRAM entry. Also, we assume that there are M entries in the memory, w_k is the key width and w_v is the data value width. The number of BRAMs consumed by a generic dMHC(k,c) instance for implementing the match portion of the memory can then be expressed as:

$$BRAM_{dmhc\_match} = k \times \left\lceil \frac{w_k + w_v}{32} \right\rceil \times \left\lceil \frac{cM}{1024} \right\rceil \qquad (8)$$

dMHC needs to perform logic computation in form of hash function computations, XOR-reduce on the G table outputs and the final match on the key. The number of LUTs needed for these are expressed in Table 2.

Table 2: 6-LUTs consumed by dMHC(k,c)

Hash Functions and XOR-reduce	Key Match
$k \times \log_2(cM) \times \left\lceil \left(\left\lceil \frac{w_k}{\log_2(cM)} \right\rceil + 1 \right) /6 \right\rceil$ $+(w_k + w_v) \times \left\lceil \frac{k}{6} \right\rceil$	$\left\lceil \frac{w_k}{3} \right\rceil$ $+ \left\lceil \frac{\lceil \frac{w_k}{3} \rceil}{5} \right\rceil$

Since BRAMs are scarcer than LUTs, we can understand most of the benefits by comparing BRAM usage for a fully associative memory's match and the G tables in a generic dMHC(k,c) design. Revising Eq. 1 to use the same parameters as our dMHC(k,c) area model:

$$BRAM_{fully\_assoc\_match} = \left\lceil \frac{w_k}{10} \right\rceil \times \left\lceil \frac{M}{32} \right\rceil \qquad (9)$$

Taking the ratio of these BRAM counts, we get:

$$\frac{BRAM_{dmhc\_match}}{BRAM_{fully\_assoc\_match}} = \frac{k \times \lceil \frac{w_k + w_v}{32} \rceil \times \lceil \frac{cM}{1024} \rceil}{\left(\lceil \frac{w_k}{10} \rceil \times \lceil \frac{M}{32} \rceil \right)} \approx \frac{kc}{100} \times \frac{w_k + w_v}{w_k} \qquad (10)$$

In case $w_v \approx w_k$, we can reduce the above expression to:

$$\frac{BRAM_{dmhc\_match}}{BRAM_{fully\_assoc\_match}} \approx \frac{kc}{50} \qquad (11)$$

From this we can observe that, in case $k = 4, c = 2$ suffices, *the* dMHC*(4,2) match unit uses less than one-sixth the BRAMs of the fully associative memory (for $w_k \approx w_v$).*

3.6 Reducing G-Table Width

The G table architecture as described in the previous sections provides the same functionality as the exhaustive search in a fully associative memory's matrix, albeit with a low (configurable in k and c) conflict rate. Each entry in our G tables is comprised of a w_k-bit wide *key* field and a w_v-bit wide *value* field. This could directly translate into a very wide G table whenever the key is wide and/or the data value is wide. On top of this, our architecture has to store these fields k times for k hash functions. This is primarily because, given an input key, we are trying to match the key as well as fetch the data value in a single BRAM cycle as shown in Fig. 2. For the rest of the paper, we refer to this design as the Flat dMHC design.

In the ideal case, we would like to only keep a single copy all the key-value pairs (instead of k copies). We can modify the Flat dMHC architecture to do just that. The simple idea is that we store all the key-value pairs only once in a single table and only store their address information in the G tables. Then, given a key, we can fetch these k G table entries and XOR them together to get the exact memory

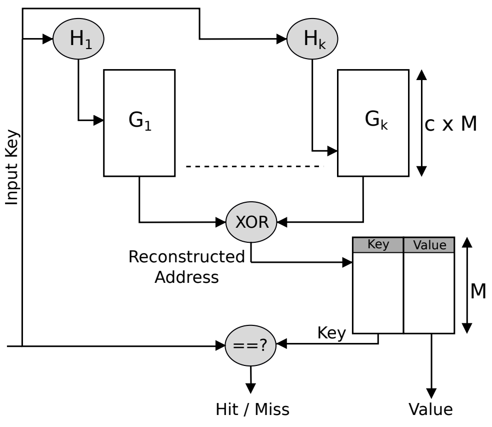

Figure 4: A Two-Level dMHC(k,c)

location of the key in the first BRAM cycle. Then, in the second BRAM cycle, we can fetch the key-value pair from that location and perform the match on the key to rule out a false-positive. The resulting dMHC architecture is shown in the Fig. 4. As we can see in the figure, this new design results in a 2 BRAM cycle access, hence, we call it the 2-level dMHC. The two cycle access with a level of indirection is similar to the perfect hash design in [7].

For a dMHC with M entries, the addresses are of the order $\log_2(M)$. Therefore, the BRAM consumption for the G tables falls from $O((w_k + w_v) \times M)$ in case of the Flat dMHC to $O(M \log_2(M))$ for the 2-level dMHC for any (k, c). This can result in significant reduction in BRAM consumption for the G tables as the 2-level dMHC G table widths are independent of the widths the key-value pairs.

Modifying Eq. (8) for the 2-level dMHC design, we get:

$$BRAM_{dmhc\_match\_2level} = k \times \left\lceil \frac{\log_2(M)}{32} \right\rceil \times \left\lceil \frac{cM}{1024} \right\rceil \quad (12)$$

Taking the ratio of the BRAMs consumed for the match unit in the 2-level dMHC against the fully associative match, we get:

$$\frac{BRAM_{dmhc\_match\_2level}}{BRAM_{fully\_assoc\_match}} = \frac{k \times \left\lceil \frac{\log_2(M)}{32} \right\rceil \times \left\lceil \frac{cM}{1024} \right\rceil}{\left(\left\lceil \frac{w_k}{10} \right\rceil \times \left\lceil \frac{M}{32} \right\rceil \right)}$$
$$\approx \frac{kc}{100} \times \frac{\log_2(M)}{w_k} \quad (13)$$

In comparison to the Flat dMHC design, the 2-level dMHC design provides additional BRAM savings as long as $\log_2(M) < 2w_k$. In a typical case, where w_k is 64-bits, we save BRAMs as long as our capacity is less than 2^{128} entries, which is much larger than one would expect to see in practice. Now, *for the 2-level* dMHC *with* $w_k = 64$ *bits, a* dMHC*(4,2) with 1024 entries would consume* $\frac{1}{80}^{th}$ *of the BRAMs consumed by the fully associative memory* — roughly 14× less than the flat dMHC design.

3.6.1 A Performance-Area Hybrid dMHC

The Flat dMHC gives us a single BRAM cycle latency but consumes a large number of BRAMs. The 2-level dMHC consumes significantly fewer BRAMs, but, results in a two BRAM cycle access. Even for the latency sensitive cases, there could be two cases: (1) where we need to know if the key-value is present in the memory as soon as possible, or, (2) where we need the data value quickly, and we can confirm the presence in the memory later (*e.g.* in a processor pipeline where we can squash the operation in later pipeline stages). It is possible to modify our 2-level dMHC to achieve both these cases. For (1), we can simply add the key fields back into the G tables. This will allow us to reconstruct the key in the first BRAM cycle and signal the rest of the system if it is found in the memory or not. For (2), we need to add the data value fields in the G tables and then we can simply reconstruct the value in the first BRAM cycle.

3.7 Minimum Area to Achieve Miss-Rate

Let us assume a dMHC of M entries with w_k-wide keys and w_v-wide data values. Ideally, there may be multiple ways to achieve a particular conflict rate since there could be multiple (k, c)-pairs that achieve the same conflict miss probability (see Eq. (6)). Thus, it should be possible to choose the BRAM-optimal dMHC configuration to achieve a given conflict miss probability for a given memory capacity.

Since the parameter c should be a power of 2, let $c = 2^g$. Also, from Eq. (6), we have:

$$P_{conflict} \approx \frac{1}{c^k} = 2^{-kg} \quad (14)$$

In order to achieve an arbitrarily low conflict probability, we can equate the above expression to a low value, say,

$$2^{-kg} = 2^{-n} \text{ or, } kg = n \quad (15)$$

For example, $n = 16$ gives a conflict miss probability of 1 in 65536. With $n = 16$, we have the options in implementing a dMHC with $(g = 1, k = 16)$ to $(g = 16, k = 1)$. We can make this decision based on the number of BRAMs consumed for each of the above configurations. For this we only consider the number of BRAMs consumed by the match unit (i.e., G tables). We start with Eq. 8. Since, the G table width ($w_k + w_v$ in the flat case in Eq. 8 or $\log_2(M)$ in the 2-level case) is independent of k and c, we can replace it with a constant α. As we will see, the final result is independent of α, so the conclusion here holds for all dMHC variants.

$$BRAM_{dmhc\_match}(k, c) = \alpha \times k \times \left\lceil \frac{cM}{1024} \right\rceil$$

Now, $k = \frac{n}{g} = \frac{n}{\log_2(c)}$. Therefore,

$$BRAM_{dmhc\_match}(c) \approx \alpha' \times \frac{ncM}{\log_2(c)} \quad (16)$$

Taking derivative of Eq. 16 w.r.t c, we see that it is minimized for $c = e$ (=2.718). Since we demand that c be a power of 2, that suggests the best choice is to always set c to 2 or 4. Later in Sec. 5, we experimentally show that $c = 2$ is sufficient to achieve a near-associative performance.

4. dMHC MEMORY MANAGEMENT

To manage an M-deep dMHC dynamically, holding at most M match values at a time, we will need to delete and insert values in the memory and occasionally relocate values.

- If we are at capacity, we need to select an entry and remove it from the memory. This involves some cleanup of state (Sec. 4.3).
- Once we have space, we need to insert the new entry into the memory.

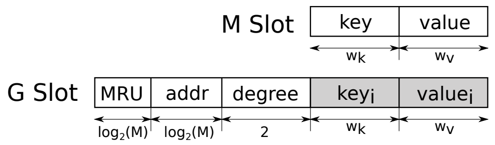

Figure 5: G and M slot compositions for a dMHC with capacity M, w_k-wide keys and w_v-wide values. The shaded fields in the G slots are only present in the Flat dMHC design (MRU: Most Recent User)

1. most of time $(1-(1-e^{-\frac{1}{c}})^k)$, this simply means writing values into memories (Sec. 4.4).
2. a small fraction of the time ($< c^{-k}$), the new entry will conflict with all k existing entries.
   (a) In most conflict cases ($> 1-(2c^2(e^{\frac{1}{c}}-1))^{-k}$) one or more of the conflicting G-table users will only be used by a single, existing entry (Sec. 4.5). By removing one of these entries, we can insert the new entry.
   (b) We can then try to re-insert the entry we just removed. With probability $> 1-c^{k-1}$, we can re-insert this entry without conflict (Sec. 4.5).
   (c) If it conflicts, we continue removing and re-inserting entries with similar probability of success. As a result, we can almost always eventually accommodate all the entries in the memory (Sec. 4.6).

The remainder of this section describes the details of the state and operations needed to implement our management algorithm.

4.1 Table Composition

We refer to each entry in a G table as a G slot, and the number of key-value pairs using a particular G slot as its degree. The table with the original key-value pairs is the M table, and we refer to each entry in there as an M slot. Fig. 5 shows the composition of each G slot and M slot. The remainder of this section explains the rationale and use for each of the subcomponents of these table entries.

4.2 Servicing Misses in dMHC

In the dMHC architecture, like in an associative memory or any cache, a miss occurs when the input data is not found in the memory. In the dMHC architecture, we could have a compulsory miss, capacity miss or a conflict miss (on the other hand, an associative memory has no conflict misses). Upon a miss, in order to insert the new key-value pair into the memory, the first step is to find space in the memory for insertion. For a capacity M dMHC we cannot hold more than M key-value pairs at a given time. If there are less than M key-value pairs stored in the memory, then we have empty slots for inserting the new key-value pair. However, if we are already at capacity, we need to evict a key-value pair in order to accommodate the incoming key-value pair.

There exist many eviction policies such as Least Recently Used (LRU) and Least Frequently Used (LFU). For our dMHC architecture, we use the Least Recently Inserted (LRI) policy. In most cases LRI policy performs as well as the LRU policy but requires less state to be maintained (LRU requires keeping age for each entry, where as LRI can be implemented simply as a single global counter). In Sec. 4.5 we further highlight the advantage of using the LRI policy in the dMHC architecture.

4.3 Clean-up on Eviction

As explained in Sec. 3, each key-value pair is stored by assigning suitable values to the G slots hashed into by the key. Moreover, the conflict probability computation in Eq. 6 assumed that, for a maximum of M key-value pairs in the memory, no more than M G slots (out of a total of $c \times M$) are being used in the G tables. Assuming uniformly distributed hash functions, the used G slots are uniformly distributed. When we are evicting a key-value pair, if we do not cleanup the G slots being used by the evicted key-value pair, then we could potentially end up in a situation where there are more than M G slots in use in one or more G tables, which would increase the conflict probability computed in Eq. 6. Therefore it is necessary to free up the G slots that are not being used for storing the key-value pairs present in the memory in order to continue reaping the benefits of the low probability as given by Eq. (6). Cleaning up a G slot simply requires resetting its contents to all zeros. At the same time, it is possible, albeit with a low probability, that a G slot used by the evicted key-value pair was being used by another key-value pair still present in the memory. In that case, we do not want to reset the contents of that G slot, because it would render that other key-value pair unreachable, effectively evicting it from the memory.

In order to solve this problem, we store the *degree* of each G slot along with the key-value information. This is the same basic solution used to allow deletion in counting Bloom filters [8]. Now, we can only reset those G slots that have a degree *one*, as they were being used exclusively by the evicted key-value pair. We also decrement the degree of all other G slots, as now they are being used by one less key-value pair. For an M-deep dMHC, the maximum degree of a G slot could be M, adding $\log_2(M)$ bits to the G slot. However, with a high sparsity factor and uniformly random hash functions, the maximum expected value of the degree is low. For example, at any given time, with proper cleanup, probability of all G slots being used by two or more key-value pairs is close to $(2c^2(e^{\frac{1}{c}}-1))^{-k}$, which is 0.14% for a dMHC(4,2). In order to corroborate this analytical result, we also simulated a dMHC(4,2) for a set of SPEC2006 benchmark programs and recorded the degree of G slots for each eviction. For k=4, c=2, M=1024, the average degree is 1.01, and the probability the degree is 2 or greater is less than 0.007%. Consequently, we can get away with using a small number of bits in the G slot for keeping track of its degree (2 bits in our current implementation). Although uncommon, the degree of a G slot can overflow the maximum of three in our designs. The only consequence of this, is that we may end up freeing the slot prematurely, forcing us to take a miss to refill the slot.

4.4 Inserting data into dMHC

Once we have free space in our memory, we can go ahead and insert the new key-value pair. The new key hashes into k G slots. With a high probability of $1-(1-e^{-\frac{1}{c}})^k$ (0.976 for a dMHC(4,2)), the G slots hashed into by the new key will not all be in use by the key-value pairs already present in the memory. In other words, with a high probability we can find at least one degree zero G slot which is not being used to store any key-value pair. Then, we can assign that G slot suitable values (all the fields) such that all the k G slots can now reproduce the original key-value pair for the Flat dMHC design or the location in the memory for the 2-level dMHC design. This requires the same XOR calculations as shown in Eq. (4). At the same time, we increment the degree of all the G slots used by the new key-value pair.

4.5 Resolving Conflicts in dMHC

With a probability roughly equal to $(1-e^{-\frac{1}{c}})^k$ (0.024 for a dMHC(4,2) design), all the G slots hashed into by the incoming key will be in use by one or more key-value pairs already present in the memory. In that case, we will have to re-assign the fields in at least one G slot in order to accommodate the new key-value pair. Since all of these G slots are being used by other key-value pairs, re-assigning their values will render the associated key-value pairs unreachable, effectively evicting them from the memory due to this newly created conflict (we call them being victimized). Nevertheless, in order to be able to insert a new key-value pair, we must re-assign values in at least one G slot.

Mathematically, whenever such a conflict occurs, we can find a G slot that is being used by only a single key-value pair with a probability greater than $1-(2c^2(e^{\frac{1}{c}}-1))^{-k}$ (0.9986 for dMHC(4,2)). Once we are able to locate a G slot that has a degree of one, we can re-assign its fields such that the new fields, along with the fields in the other $k-1$ G slots, correspond to the newly inserted key-value pair.

By re-assigning the G slot fields we victimize one or more existing key-value pairs, one in the most common case. However, since each key-value pair is stored using k, G slots, it might be possible to re-insert a victimized key-value pair by modifying the fields in another of its remaining $(k-1)$ G slots. Continuing the idea of Eq. (6), with a probability of $1-c^{1-k}$, we can re-insert this entry by modifying a G slot which is being used only by this key-value pair. Here the conflict probability is c^{1-k} rather than c^{-k} because we know it will conflict with the newly inserted entry that caused the this key-value pair to be victimized in the first place. However, with a very low probability $((2c^2(e^{\frac{1}{c}}-1))^{-k})$, we create another conflict (when the G slot chosen to re-insert the victimized key-value pair has a degree greater than one). In that case, we continue removing and re-inserting entries with similar probability of success. As a result, we can almost always eventually accommodate all the entries in the memory, resulting in a generalized *N-hop Repair* strategy, where at each hop we re-insert a victimized key-value pair. This is equivalent to moving a hash entry to accommodate an insertion (*c.f.* [11]).

In order to be able to evict and re-insert the key-value pairs, we need to store all the original key-value pairs as well; this allows us to recompute the hash values and the new values to be assigned to the G slot fields. The 2-level dMHC is already storing these key-value pairs, but this forces us to add an M table for the Flat dMHC. Furthermore, to repair the victimized key-value pair, we need the address of the M slot it is stored in. Therefore, we add another $\log_2(M)$ bits to a G slot giving us the address of that key-value pair that used this G slot most recently. This way we only repair the key-value pair that was accessed most recently using this G slot (we do not expect this G slot to be used by more than one key-value pairs in the most common case).

When we do victimize more than one key-value pairs (less than 0.14% of the time for dMHC(4,2)) two things go bad – (a) since we only re-insert one of the victimized key-value-pairs, we lose memory capacity by letting the other victimized key-value pairs stay in the memory even though they cannot be accessed anymore, and (b) the G slots storing information for these key-value pairs are not cleaned up as explained in the Sec. 4.3, affecting the conflict miss probability. However, since the LRI policy chooses the M slot to be evicted in a periodic manner, we will eventually be able to evict these stale key-value pairs and also cleanup their G slots.

```
**Function** LDVN(K, N)
// $K$ is the input key
$h_i \leftarrow H_i(K)$ for $1 \le i \le k$
$key \leftarrow$ XOR$(G_i[h_i].key ... G_k[h_k].key)$
$val \leftarrow$ XOR$(G_i[h_i].val ... G_k[h_k].val)$
$m \leftarrow$ XOR$(G_i[h_i].addr ... G_k[h_k].addr)$
**if** $key{==}K$ **then**
    $G_i[h_i].m \leftarrow m$ for $1 \le i \le k$
    **return** $val$  // hit
**else**
    $new\_m \leftarrow m\_slot\_counter$
    $h_i^{'} \leftarrow H_i(\mathrm{M}[new\_m].key)$ for $1 \le i \le k$
    cleanup$(G_i[h_i^{'}])$ for $1 \le i \le k$
    **if** there is an unused $h_i$ **then**
        use $h_i$ to store $K$ at M$[new\_m]$
    **else**
        choose $i$ s.t. deg$(G_i[h_i])$ =
                min_deg$(G_1[h_1] ... G_k[h_k])$
        re-assign $G_i[h_i].\{key,\ val,\ addr\}$ to store $K$ at
        M$[new\_m]$
        $victim\_m \leftarrow G_i[h_i].m$
        LDV$(victim\_m,\ i,\ N)$;
    **end**
    $G_i[h_i].m \leftarrow new\_m$ for $1 \le i \le k$
    $m\_slot\_counter++$ // LRI replacement of M slots
    **return** dMHC_miss
**end**
**EndFunction**

**Function** LDV$(m\_slot,\ j,\ N)$
**if** $N = 0$ **then**
    **return** 0  //no more hops
**else**
    $h_i \leftarrow H_i(\mathrm{M}[m\_slot].key)$ for $1 \le i \le k$
    choose $i \ne j$ s.t. deg$(G_i[h_i])$ =
            min_deg$(G_1[h_1] ... G_k[h_k])$
    re-assign $G_i[h_i].\{key,\ val,\ addr\}$ to store
            M$[m\_slot]$.key at M$[m\_slot]$
    $victim\_m \leftarrow G_i[h_i].m$
    $G_i[h_i].m \leftarrow m\_slot$
    **return** LDV$(victim\_m,\ i,\ N-1)$
**end**
**EndFunction**
```

Algorithm 1: Pseudocode for Lowest Degree Victim with N-hop Repair Policy

4.6 Lowest Degree Victim with N-hop Repair

Generalizing the strategy above, this brings us to the Lowest Degree Victimization policy for inserting new key-value pairs in case of a conflict: to resolve a conflict, we reassign the G slot with the lowest degree which would victimize the

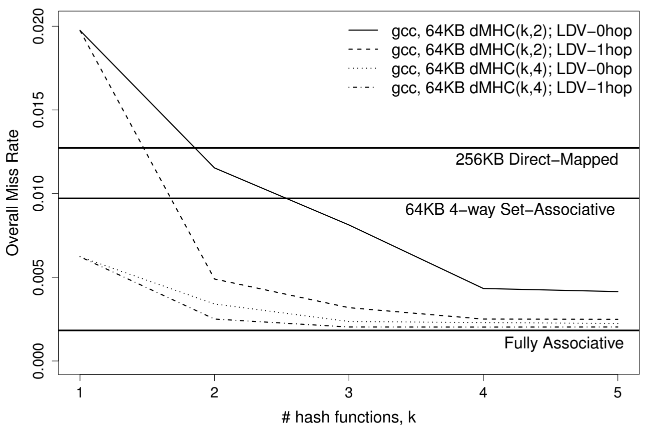

Figure 6: Miss-Rates for a 64KB dMHC for gcc

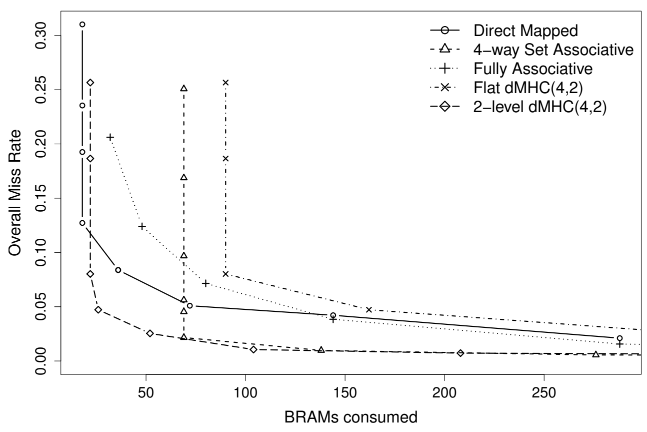

Figure 7: BRAMs v/s Miss-Rates for gcc

least number of M slots. Algorithm 1 shows the complete algorithm for dMHC memory management.

5. PERFORMANCE COMPARISON

5.1 Case Study: L1 Data Cache Miss-Rates

Fully associative memories would make for high performance L1 data (or instruction) caches for a processor, albeit with heavy area overheads. The large overhead is why we do not see them as on-chip caches in a commodity processor. Our analytical model shows that the dMHC architecture can achieve a near-associative memory performance at much lower BRAM consumption (Sec. 3). To validate our theoretical performance and area predictions, we modeled the dMHC as an L1 data cache and performed address trace-driven simulations on a small set of eight benchmark programs from the SPEC2006 Benchmark Suite [9] using traces from a 64-bit x86-simulator [2] simulating each benchmark for 100M cycles. Memory reference counts for the address traces used in the present work are highlighted in Table 3 (column I).

In order to perform a direct comparison, we also simulated a fully associative memory and several set-associative caches for the same benchmarks. Fig. 6 shows how the overall miss-rate varies for our architecture with respect to the parameters k and c for the benchmark gcc for a 64KB L1 dMHC cache with a block size of 8, 64b data values. (miss-rate is same for both Flat and 2-level dMHC designs). The figure also shows the miss-rate achieved with a fully associative memory of same capacity as the dMHC, a direct-mapped cache with four times the capacity and a 4-way set-associative cache of same capacity. As suggested by our analytical model, increasing the values of k and/or c reduces the number of conflicts (thereby reducing the overall miss-rate), approaching the miss-rate achieved by a fully associative memory of the same capacity at high values. Moreover, some dMHC configurations perform better than a bigger direct-mapped cache and a set-associative cache of same capacity. Also, the 1-hop repair strategy outperforms the 0-hop strategy for the same dMHC configurations. In Sec. 5.3, we compare the BRAM consumption for these caches.

5.2 Hardware Implementation

We implemented the proposed dMHC architecture (both Flat and 2-level designs) in Bluespec SystemVerilog [4] hardware description language. Our tool[1] can generate a parameterized dMHC instance to target a particular conflict miss-rate or BRAM budget. Using the Bluespec compiler we generate Verilog HDL code which we then synthesize using Xilinx ISE 13.2 toolchain. We also implemented the 0-hop and 1-hop LDV policies for memory management directly in Bluespec as low level control FSMs. In order to reduce the miss-service latency in the memory controller, we have implemented both the policies as parallel as possible.

5.3 Case Study: L1 Data Cache BRAMs

Table 3 shows the BRAM usage ratio for eight SPEC2006 benchmarks for a 64KB L1 data cache. For each benchmark, we identify a dMHC instance that uses the least number of BRAMs while achieving a near-associative miss rate (that is, less than 5% of misses are due to conflicts). In each row we indicate the conflict ratio (as defined in Eq. (7)) and the most BRAM-efficient dMHC configuration achieving that. For each chosen configuration, we also report the fully associative to dMHC BRAM usage ratio (Flat and 2-level both with LDV-1hop policy). From the data in Table 3, we observe that a dMHC(4,2) configuration with 1-hop repair policy is able to achieve desirable conflict ratios in most of the cases.

Results from Table 3 show that our architecture is able to achieve a near-associative performance for a dMHC(4,2) configuration. However, it is also necessary to compare the BRAM cost of these designs. In order to achieve that, we extend our simulations by integrating the achieved miss-rates with the BRAM consumption for our designs as well as other caches. For this we ran simulations varying the size of all the caches from 1KB to the point where we saturate all the BRAMs available on the xc6vlx240t-2 device, and for each cache size, we record the miss-rate achieved and the number of BRAMs consumed. Fig. 7 shows how the miss-rate falls when we increase the capacity of these caches in terms of BRAMs. For any type of cache, increasing the number of BRAMs increases capacity, and therefore reduces misses. From Fig. 7, we can establish that the 2-level dMHC design is able to yield the lowest miss-rate per unit BRAM consumption across a large range of cache sizes.

Furthermore, dMHC architecture is able to achieve higher BRAM savings as the match width is increased. Fig. 9 shows

[1]http://ic.ese.upenn.edu/distributions/dmhc_fpga2013

Table 3: Fully Associative to dMHC BRAM Ratio for a 64KB L1-Cache for SPEC2006 Benchmarks

Benchmark (Mem instrs)	Conf. Ratio (%)	dMHC config (k,c)	Flat BRAM Ratio	2-level BRAM Ratio
art (19.9M)	2.6	(4,2)	0.9	6.0
gcc (15.7M)	3.2	(4,2)	0.9	6.0
go (35.5M)	3.8	(4,2)	0.9	6.0
hmmer (41.9M)	1.4	(3,2)	1.2	6.5
libq (30.2M)	0.04	(2,2)	1.7	7.2
mcf (32.2M)	2.8	(4,2)	0.9	6.0
sjeng (26.9M)	4.9	(4,2)	0.9	6.0
sphinx3 (33.1M)	4.6	(3,2)	1.2	6.5

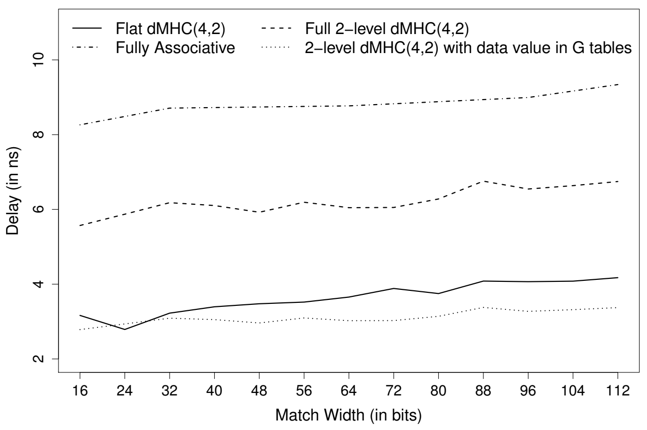

Figure 8: dMHC vs Fully Associative delays for a 1024-entry memory holding 64b values

the BRAM savings for the Flat as well as the 2-level dMHC designs over the Coregen-style fully associative memory, all of depth 1024 entries. For the Flat variant, the saving ratio saturates at about 11× where as the 2-level variant is able to save upto 100× over the fully associative memory.

5.4 dMHC Timing

Another disadvantage of the Coregen-style fully associative memory is the low frequency of operation. Reviewing Fig. 1, a fully associative memory with capacity M has an M-bit, 1-hot to $\log_2(M)$-bit dense encoder in the critical path resulting in a high latency, even when M is moderately high (say 1024). By storing the address information in the compact form ($\log_2(M)$ for a capacity of M), dMHC avoids such a slow path. In order to compare the timing performance of the dMHC architecture with the fully associative memory, we created 1024-entry dMHC and fully associative memory designs with 64b values and varying key-widths from 16b to 120b. These designs were then placed and routed using Xilinx ISE 13.2 for a Virtex 6 (xc6vlx240t-2) device. Fig. 8 shows the best-case latency achieved for these designs against the key-widths (these are the delays between providing the match key and receiving the corresponding data value out). Along with different BRAM footprints, the Flat and the 2-level dMHC designs have slightly different critical paths. Apart from that, the 2-level dMHC

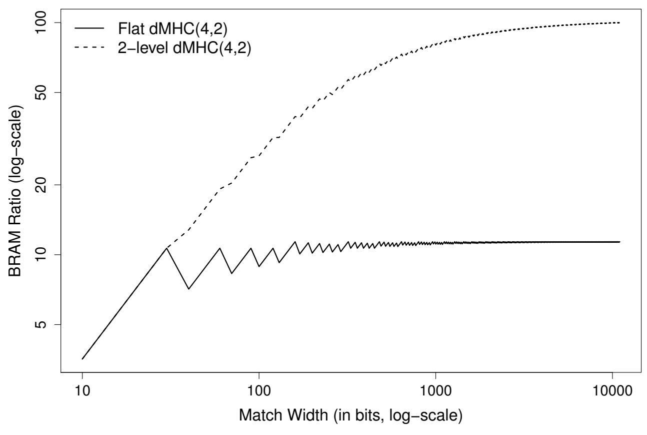

Figure 9: Fully Associative Memory to dMHC(4,2) BRAM usage ratio

requires 2 BRAM cycles to fetch the data value in the most general case. Using the 2-level dMHC variant where we store the data values in the G tables, we can achieve a much lower (single BRAM cycle) latency in the most common hit case.

6. RELATED WORK

Seznec [17] introduced a cache based on the multiple hash idea. He showed that using a cache with multiple physical ways, where each way is indexed by a different hash function, called a skewed-associative cache, resulted in a lower miss-rate than a regular direct-mapped or a set-associative cache. He further showed that a 2-way skewed-associative cache has a miss-rate close to a regular 4-way set-associative cache, however with the hardware complexity of a 2-way set-associative cache. Once we have a design that has choice, we can further reduce the conflicts by moving entries in the cache when conflicts arise [11]. Sanchez's Z-Cache extended the skewed-associative caches by introducing smart replacement policies that try to reduce the miss-rates by exploiting moves to expand the pool of eviction candidates and then choosing a suitable cache block to be evicted [16]. In the Z-Cache, there is *always* a conflict on insertion, and the question is which present entry should be removed. In most cases the dMHC has no conflict on insertion. Furthermore, since we keep track of sharing degrees, we can greedily search along a single conflicting entry for replacements, whereas the Z-Cache must expand a tree of exponentially increasing candidates. Since the Z-Cache is set associative, it demands a comparison and mux selection in the critical path after memory lookup, whereas our Flat dMHC produces the candidate result after a single memory lookup.

Bloomier filters [6] extend Bloom filters by giving the exact pattern that matched along with the set membership. These have been effectively used in applications such as accelarating virus detection using FPGA hardware [10], however setting up a Bloomier filter requires some level of preprocessing making it much more suitable for use where static support is involved. Our design has some similarity to Song's multiple hash function counting Bloom Filter [19]. However, note that Song only uses the hash function to determine the size of hash buckets that are stored off chip—particularly to avoid off-chip lookups on most cases and minimize lookups in others. Furthermore, our management logic is simpler and suitable to direct hardware implementation.

7. CONCLUSIONS

We have introduced the `dMHC` memory architecture that achieves nearly associative memory performance. Furthermore, we have shown how it can be parameterized (capacity, k, c, flat/2-level/hybrid, 0-hop/1-hop) and tuned so we can engineer the BRAM usage, conflict miss rate, and access latency of the memory. We showed that `dMHC` instances use their BRAMs more effectively than traditional alternatives (fully associative, set-associative, direct-mapped) achieving lower miss rates than the alternatives over a larger range of BRAM budgets (Sec. 5.3). Furthermore, we've shown that the `dMHC` implementations have lower access latency (Fig. 8). The `dMHC` should be in any FPGA application or reconfigurable computing designer's arsenal of building blocks.

8. ACKNOWLEDGMENTS

This material is based upon work supported by the DARPA CRASH program through the United States Air Force Research Laboratory (AFRL) under Contract No. FA8650-10-C-7090. The views expressed are those of the authors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

9. REFERENCES

[1] Y. Azar, A. Z. Border, A. R. Karlin, and E. Upfal. Balanced allocation. In *Proc. ACM STOC*, pages 593–602, 1994. 1, 3.1

[2] S. Battle, A. D. Hilton, M. Hempstead, and A. Roth. Flexible register management using reference counting. In *Proc. Intl. Symp. on High-Perf. Comp. Arch.*, pages 273–284. IEEE, 2012. 5.1

[3] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, July 1970. 3.1

[4] Bluespec, Inc. Bluespec SystemVerilog. 5.2

[5] S. Bunton and G. Borriello. Practical dictionary management for hardware data compression. *CACM*, 35(1):95–104, 1992. 2.1

[6] B. Chazelle, J. Kilian, R. Rubinfeld, and A. Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *Proc. ACM-SIAM SODA*, SODA '04, pages 30–39, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics. 6

[7] Z. J. Czech, G. Havas, and B. S. Majewski. An optimal algorithm for generating minimal perfect hash functions. *Information Processing Letters*, 43(5):257–264, 1992. 1, 3.1, 3.2, 3.6

[8] L. Fan, P. Cao, J. Almeida, and A. Z. Border. Sumary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Networking*, 8(3):281–293, 2000. 4.3

[9] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006. 5.1

[10] J. Ho and G. Lemieux. PERG: A scalable FPGA-based pattern-matching engine with consolidated bloomier filters. In *ICFPT*, pages 73–80, December 2008. 6

[11] A. Kirsch and M. Mitzenmacher. The power of one move: Hashing schemes for hardware. *IEEE/ACM Trans. Networking*, 18(6):1752–1765, 2010. 1, 4.5, 6

[12] C. E. LaForest and G. Steffan. Octavo: an FPGA-centric processor family. In *FPGA*, pages 97–106, 2012. 1

[13] S.-L. L. Lu, P. Yiannacouras, T. Suh, R. Kassa, and M. Konow. A desktop computer with a reconfigurable Pentium. *ACM Tr. Reconfig. Tech. and Sys.*, 1(1), March 2008. 1

[14] M. Mitzenmacher. Studying balanced allocation with differential equations. *Combinatorics, Probability, and Computing*, 8(5):473–482, 1999. 1, 3.1

[15] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown. Implementing an OpenFlow switch on the NetFPGA platform. In *Proc. ACM/IEEE Symp. ANCS*, pages 1–9, 2008. 2.1

[16] D. Sanchez and C. Kozyrakis. The ZCache: Decoupling ways and associativity. In *MICRO*, pages 196–207, 2010. 1, 6

[17] A. Seznec. A case for two-way skewed-associative caches. In *ISCA*, pages 169–178, 1993. 6

[18] A. Seznec and F. Bodin. Skewed-associative caches. In *PARLE*, pages 304–316, 1993. 3.2

[19] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. In *Proceedings of the Conference on Applications, technologies, architectures, and protocols for computer communications*, pages 181–192, 2005. 6

[20] J. Wawrzynek, D. Patterson, M. Oskin, S.-L. Lu, C. Kozyrakis, J. C. Hoe, D. Chiou, and K. Asanović. RAMP: Research accelerator for multiple processors. *IEEE Micro*, 27(2):46–57, 2007. 1

[21] S. Wee, J. Casper, N. Njoroge, Y. Tesylar, D. Ge, C. Kozyrakis, and K. Olukotun. A practical FPGA based framework for novel CMP research. In *FPGA*, pages 116–125, 2007. 1

[22] R. Wunderlich and J. C. Hoe. In-system FPGA prototyping of an Itanium microarchitecture. In *ICCD*, pages 288–294, 2004. 1

[23] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *Parameterizable Content-Addressable Memory*, March 2011. XAPP 1151 <`http://www.xilinx.com/support/documentation/application_notes/xapp1151_Param_CAM.pdf`>. 1, 2.2

[24] Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. *Virtex-6 FPGA Data Sheet: DC and Switching Characteristics*, September 2011. DS512 <`http://www.xilinx.com/support/documentation/data_sheets/ds152.pdf`>. 2.2

[25] P. Yiannacouras and J. Rose. A parameterized automatic cache generator for FPGAs. In *ICFPT*, pages 324–327, 2003. 1

[26] P. Yiannacouras, J. G. Steffan, and J. Rose. Exploration and customization of FPGA-based soft processors. *IEEE Trans. Computer-Aided Design*, 26(2):266–277, 2007 2007. 1

APPENDIX

A. PDF OF DEGREES

The problem of expressing the probability distribution of the degrees of the G slots can be modelled as the Birthday Problem.[2] In a generalized Birthday Problem, one needs to find the probability that in a group of N people, n have the same birthday (any particular date). Although an exact answer can be found using the binomial distribution, this probability can be nicely approximated by the Poisson distribution as follows:

$$P(n) = e^{-\lambda} \frac{\lambda^n}{n!} \tag{17}$$

where, λ is the expected value of the number of birthdays on a single day ($\lambda = \frac{N}{365}$).

In the case of dMHC, the problem is that given cM slots in a G table (equivalent to the number of days in a year), what is the probability that exactly d key-value pairs will use a particular G slot (defined earlier as its degree) when M key-value pairs (equivalent to the number of people) are being stored in the dMHC. Therefore, in this case,

$$\lambda = \frac{M}{cM} = \frac{1}{c} \tag{18}$$

Now, the probability distribution for the degrees of the G slots can be expressed using the Poisson distribution as follows:

$$P(degree = d) = P(d) = e^{-\lambda} \frac{\lambda^d}{d!} \tag{19}$$

Since, the expected value of the Poisson distribution is λ, therefore, expected value of the degree of any G slot is $\lambda = \frac{1}{c}$, as expected.

While inserting a new key-value pair in a dMHC(k,c), we preferably look for a degree-0, G slot so that we do not victimize any of the existing key-value pairs. The probability that we can find at least one degree-0 G slot (out of k slots) can be expressed as:

$$\begin{aligned} P(\text{at least 1 degree-0}) &= 1 - P(\text{all non-zero degrees}) \\ &= 1 - (1 - P(0))^k \\ &= 1 - (1 - e^{-\lambda})^k \end{aligned}$$

For $k = 4, c = 2$, this is 0.974 (probability that we do not victimize any G slot while inserting a new key-value pair). If we vary the sparsity factor and the number of hash functions, we can plot the probability that we do not victimize any G slot for different cases as shown in the Figure 10.

B. DEGREE OF VICTIMS

We can extend the Poisson distribution to also express the distribution of the degrees of the victimized G slots, in case we do not find a degree-0 G slot. For this, we can find the probability that degree of a G slot is d given that it is being used to store at least one key-value pair. That is,

$$\begin{aligned} P_{victim}(degree = d) &= P(d|d \geq 1) \\ &= \frac{P(d)}{(1 - P(0))} \\ &= \frac{e^{-\lambda}}{1 - e^{-\lambda}} \frac{\lambda^d}{d!} \end{aligned}$$

[2]McKinney, EH (1966), Generalized Birthday Problem, *The American Mathematical Monthly*, 73(4): 385–387

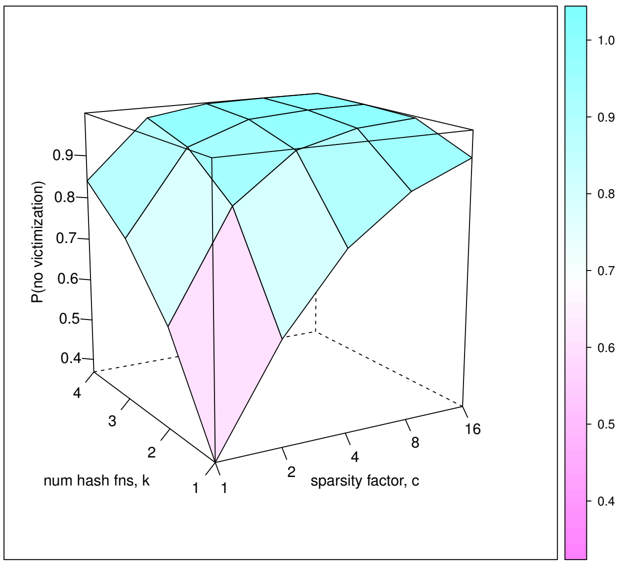

Figure 10: Probability of no victimization

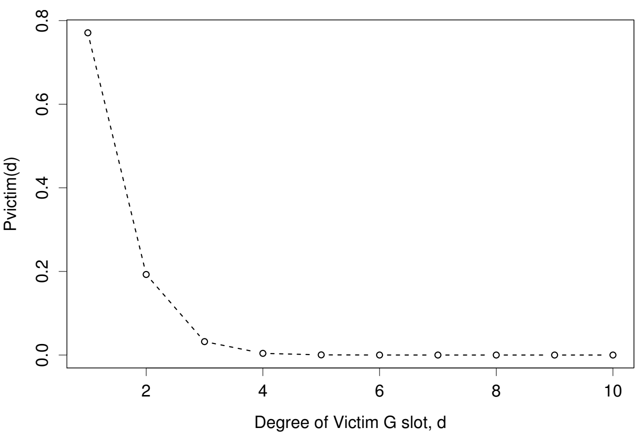

Figure 11: PDF for degrees of victims, $c = 2$

This is shown in the Figure 11. As can be seen in the figure, with a high probability we victimize only one key-value pair. We can now find the expected value of the degree of a G slot upon victimization, which comes out to 1.27.

Also, as mentioned earlier, we only keep 2 bits for maintaining the degree of G slots. That is, we saturate the degree at three. This works well if the probability that the degree of a G slot is greater than three is very low. For $k = 4, c = 2$:

$$P(d > 3|d \geq 1) = \sum_{d \geq 4} P_{victim}(d) = 0.0044 \tag{20}$$