Area-Efficient Near-Associative Memories on FPGAs

UDIT DHAWAN and ANDRÉ DEHON, University of Pennsylvania

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 article, we develop a new, FPGA-friendly, memory system architecture based on a multiple hash scheme that is able to achieve near-associative performance without the area-delay overheads of a fully associative memory on FPGAs. At the same time, we develop a novel memory management algorithm that allows us to statistically mimic an associative memory. Using the proposed architecture as a 64KB L1 data cache, we show that it is able to achieve near-associative miss rates while consuming 3–13× fewer FPGA memory resources for a set of benchmark programs from the SPEC CPU2006 suite than fully associative memories generated by the Xilinx Coregen tool. Benefits for our architecture increase with key width, allowing area reduction up to 100×. Mapping delay is also reduced to 3.7ns for a 1,024-entry flat version or 6.1ns for an area-efficient version compared to 17.6ns for a fully associative memory for a 64-bit key on a Xilinx Virtex 6 device.

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

Additional Key Words and Phrases: FPGA, BRAM, associative memory, CAM, cache, hashing

ACM Reference Format:
Udit Dhawan and André DeHon. 2015. Area-efficient near-associative memories on FPGAs. ACM Trans. Reconfig. Technol. Syst. 7, 4, Article 3 (January 2015), 22 pages.
DOI: http://dx.doi.org/10.1145/2629471

1. INTRODUCTION

With increasing use of high-frequency soft processors on FPGAs (e.g., [Yiannacouras et al. 2007; LaForest and Steffan 2012]) and an increasing use of FPGAs for processor emulation (e.g., [Wunderlich and Hoe 2004; Wee et al. 2007; Wawrzynek et al. 2007; Lu et al. 2008]), we need to be able to implement high-performance memory subsystems 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 [Wee et al. 2007] observed:

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

This material is based on work supported by the DARPA CRASH program through the United States Air Force Research Laboratory (AFRL) under Contract No. FA8650-10-C-7090.
Authors' addresses: U. Dhawan, Electrical and Systems Engineering Department, University of Pennsylvania, 3330 Walnut Street, Philadelphia, PA 19104; email: udit@seas.upenn.edu; A. DeHon, Department of Electrical and Systems Engineering, University of Pennsylvania, 3330 Walnut Street, Philadelphia, PA 19104; email: andre@acm.org.

DOI: http://dx.doi.org/10.1145/2629471

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 [Xilinx, Inc. 2011a], 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 60 BRAMs to perform the match, despite the fact that 512 sixty-four-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* 60× *the stored memory capacity.* The overheads increase with the match width. Yiannacouras and Rose [2003] show 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 [Azar et al. 1994; Mitzenmacher 1999] based on a generalization of Czech et al. [1992] that can be efficiently implemented on top of BRAMs. We further develop efficient replacement policies exploiting the power of choice [Azar et al. 1994; Mitzenmacher 1999; Sanchez and Kozyrakis 2010; Kirsch and Mitzenmacher 2010]. This allows us to reduce the conflict miss probability to below 0.03% for the 512-entry CAM while using only six total BRAMs.

Our novel contributions include the following:

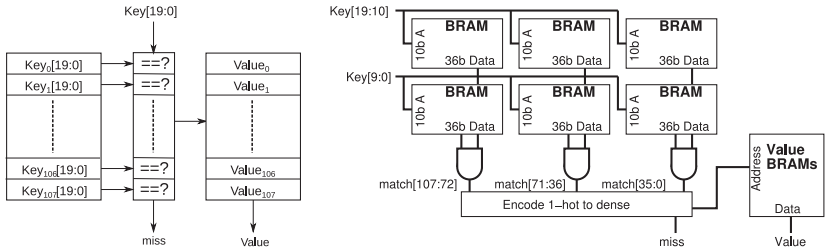
—Customization of the table-based Perfect Hash scheme [Czech et al. 1992] for efficient implementation on FPGAs (Section 3.2)
—FPGA-customized memory architecture that can be tuned to trade off BRAM usage with conflict miss rate (Section 3)
—Analytic characterization of capacity (Section 3.5) and miss rate (Section 3.3), showing that the architecture can achieve very low (≈0.05%) conflict miss rates with substantially fewer BRAMs than Xilinx Coregen-style associative memories
—Analytic derivation of optimal sparsity factor (Section 3.8)
—Identification of a family of replacement policies and characterization of their performance, area, and cycle time implications (Section 4)
—Empirical quantitative comparison of the area and performance of our new memory organization against fully associative and set-associative memories (Section 5)

This work is an expansion of Dhawan and DeHon [2013]. Extensions include analysis of false-positive rate in a `dMHC` design (Section 3.4), details of the hybrid `dMHC` variants (Section 3.7), configuring a `dMHC` instance for a target BRAM consumption (Section 3.9), area-delay characterization of the management algorithm (Section 4.7), and a limit-study experiment on the management algorithm to show its efficacy (Section 5.5).

2. BACKGROUND

2.1. Associative Memories

An associative memory provides a conflict-free 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), first-in first-out (FIFO), or least frequently used (LFU).



(a) Custom Implementation (b) BRAM Implementation

Fig. 1. Twenty-bit match, 108-entry fully associative memory.

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 can be crucial to enhance performance in many applications like network routing [Naous et al. 2008] and dictionary lookups for pattern matching and data compression/decompression [Bunton and Borriello 1992].

2.2. Fully Associative Memories on FPGAs

In a custom implementation (Figure 1(a)), an associative address-match cell is programmable so it can match against any key. Since FPGAs only contain ordinary SRAM blocks (where the address-match cell is fixed), CAMs must be built out of logic and these embedded SRAMs (e.g., BRAMs), as shown in Figure 1(b).

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 [Xilinx, Inc. 2011a] for a fully associative memory for a Virtex 6 FPGA (xc6vlx240t-2 device) [Xilinx, Inc. 2011b]. This device contains 416 thirty-six-Kbit Block RAMs, which can be organized as 2,048 $\times$ 18, 1,024 $\times$ 36, or 512 $\times$ 72 memories. In order to build an n-deep CAM with m-bit keys on a Virtex FPGA, Coregen organizes it as a matrix with 2^m rows (a row each for all the *possible* 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, 36-entry-deep CAM match unit in a single BRAM (using a 1,024 $\times$ 36 configuration) [Xilinx, Inc. 2011a]. 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 $\lceil \frac{n}{36} \rceil$ 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 an n-deep CAM with an m-bit-wide match key using this organization is

$$\#\,\text{BRAMs} = \left\lceil \frac{m}{10} \right\rceil \times \left\lceil \frac{n}{36} \right\rceil. \tag{1}$$

Table I shows that we run out of the BRAMs available on the device (shown in italicized text) while storing 64-bit values associated with 64-bit keys, even for a moderate depth memory. 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 I. BRAMs Consumed for a Fully Associative Memory with 64-Bit Match Key and 64-Bit Data Values

Depth	BRAMs Consumed For		Total BRAMs Consumed
	Key Match	Data Value	
256	56	1	57
512	105	1	113
1,024	203	2	226
2,048	399	4	403
4,096	*798*	*8*	*806*

3. DMHC: A NEAR-ASSOCIATIVE MEMORY

The Coregen-style associative memories are inefficient for three reasons:

(1) They demand dense storage of 10-bit 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, leading to higher latency.

Ideally, we would like to be able to do almost the opposite:

(1) Densely store only present key–value pairs.
(2) Densely store results (no indirection).
(3) Directly retrieve the data from a single memory lookup.

Taking these as our targets, we develop 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 with low area-time overheads.

3.1. Basic Approach

In an ideal case, we would like 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 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 the 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 [Bloom 1970], Multihash Tables [Azar et al. 1994; Mitzenmacher 1999], and Perfect Hash functions [Czech et al. 1992]: *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 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 because multiple keys happened to have set all the hash bits associated with this key.

We define the term *sparsity factor*, c, as the ratio of the depth of the memory table to the number of values stored in the table. If we use a uniformly random hash function that maps all keys to random memory entries, then the probability of a key getting a false hit in any memory is less than $\frac{1}{c}$. Now if we use k such tables, then the probability

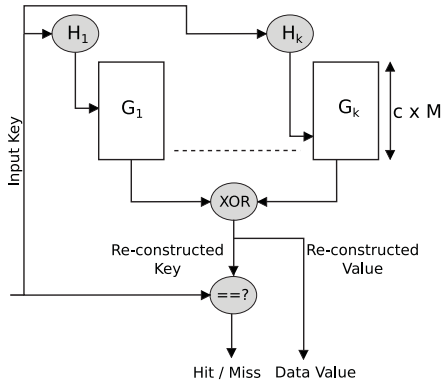


Fig. 2. A generic dMHC(k,c).

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 Section 3.8.

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 true hit.

3.2. dMHC: Hardware Organization and Operation

The top-level hardware organization of our dMHC architecture is shown in Figure 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. 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 article, we refer to a generic instance of our architecture as dMHC(k,c) with k hash functions and a sparsity factor of c.

Given an input key, we compute k, n-bit hash values, $h_1..h_k$, where $n = \log_2(c \times M)$ and M is the total number of entries in the memory. In our current implementation, we use the family of orthogonal hash functions from Seznec and Bodin [1993], which was shown to possess the properties of uniform randomness and good local dispersion. At the same time, these hash functions allow a simple FPGA implementation (see Online Appendix C). The G tables store the key–value pairs in a distributed form; that is, each key–value pair is mapped into k G table entries (each as wide as the key–value pair) that can later be combined together to form the original key–value pair. 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. The next step is to reconstruct the key–value pair. We use XOR for this purpose as shown here:

$$key = key_1 \oplus key_2 \oplus \cdots \oplus key_k \tag{2}$$

$$val = val_1 \oplus val_2 \oplus \cdots \oplus val_k. \tag{3}$$

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 Figure 2, k G table outputs are fed to an XOR-reduce tree to reconstruct the key–value pair before matching the key. In Czech et al. [1992], 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 XORs forces us to use power-of-2 G tables and M entries. In case the reconstructed key matches the input key, 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; there is a possibility of a false positive, which we discuss in Section 3.4. In case of a miss, we yield to the dMHC memory controller to service the miss, which we explain later in the Section 4.

3.3. dMHC: 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.

Since we use hash functions in our architecture, we are bound to have collisions. When a new key hashes into a G table entry that is being used by an already present key–value pair, we have a collision in that G table. The probability of an input key colliding with the present key–value pairs in a single G table is approximately

$$P_{collide} < \frac{\text{Capacity}}{\text{G Table Depth}} = \frac{|M|}{|G|} = \frac{1}{c}. \tag{4}$$

However, the dMHC makes use of multiple hash functions. Since all the hash functions are mutually orthogonal, the probability that an input key collides in all the G tables simultaneously is

$$P_{k-collide} < (P_{collide})^k \propto \frac{1}{c^k}. \tag{5}$$

As long as there is a collision in *less than* k tables, we do not qualify that as a conflict. However, a simultaneous collision in all the k tables, or a *k-collision*, is a conflict, suggesting that our conflict probability is asymptotically similar to $P_{k-collide}$. This means that by choosing high values of parameters k and c, we can make the probability $P_{k-collide}$ arbitrarily small, and hence the conflict probability. Consequently, the common case should be that new key–value pairs do not have a *k-collision* and can be inserted easily (later in Section 4 we show that even a *k-collision* might be resolved as to not result in a conflict).

We can further define the conflict miss ratio as

$$P_{conflict\_miss} = \frac{\text{Conflict Eviction Count}}{\text{Total Misses}}. \tag{6}$$

$P_{conflict\_miss}$ is zero for a fully associative memory. Figure 3 plots Equation (5) to show how the conflict miss probability falls as a function of the sparsity factor, c, for a particular number of hash functions, k. Later in Section 3.8, we see how to best choose these parameters.

3.4. False Positives in a dMHC(k,c)

As explained in Section 3.2, we compare the input key against the reconstructed key from the G tables. This can potentially result in a false positive in the unlikely event

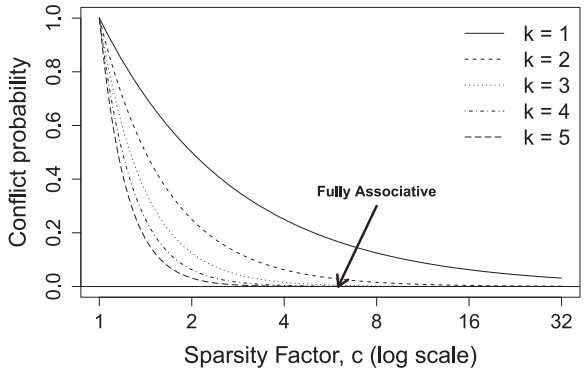


Fig. 3. dMHC conflict probabilities.

that the input key is not actually stored, but the corresponding G slots still reconstruct the correct key. There are two parts to estimating the false-positive rate: (1) if any of the G slots indexed by the input key is unused, then there cannot be a false positive since we know at once that the key has not been stored, and (2) if all the indexed G slots are being used, then there is a false positive only if the reconstructed key matches the input key, which is only as likely as 1 in 2^{-m} Therefore, for a dMHC(k,c) storing m-bit wide keys,

$$P_{false\_positive} = \left(\frac{1}{c^k}\right)\left(\frac{1}{2^m}\right). \tag{7}$$

For a dMHC configured with sufficiently large k and c parameters and a key wide enough as encountered in practical scenarios, this rate can be almost negligible—for example, storing 64-bit keys in a dMHC(4,2) leads to a false-positive rate of only 1 in 2^{68}. The false positive can be completely eliminated by performing an additional check from a separate memory storing the original key–value pairs (which, as we shall see later in Section 4, is needed for management as well) in the immediately next cycle.

3.5. dMHC Area Model

As described in the previous section, achieving near associativity with dMHC could require us to use high values of the k and c parameters. In order to 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 Section 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 $1,024 \times 36$ configuration. The model can be made more elaborate by using $2,048 \times 18$ or 512×72 configurations wherever possible to reduce BRAMs. This, however, matters only when the depth is less than 2,048 entries; beyond that, there is no benefit due to quantization effects. Also, we assume that there are M entries in the memory, key width is w_k-bit, and data value width is w_v-bit. 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

$$BRAM_{dmhc\_match} = k \times \left\lceil \frac{w_k + w_v}{36} \right\rceil \times \left\lceil \frac{cM}{1024} \right\rceil. \tag{8}$$

dMHC needs to perform logic computation in the form of hash function computations, XOR-reduce on the G table outputs, and the final match on the key. 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.[1] Revising Equation (1) to use the same parameters as our dMHC(k,c) area model,

$$BRAM_{fully\_assoc\_match} = \left\lceil \frac{w_k}{10} \right\rceil \times \left\lceil \frac{M}{36} \right\rceil. \tag{9}$$

Taking the ratio of these BRAM counts, we get

$$\frac{BRAM_{dmhc\_match}}{BRAM_{fully\_assoc\_match}} = \frac{k \times \left\lceil \frac{w_k+w_v}{36} \right\rceil \times \left\lceil \frac{cM}{1024} \right\rceil}{\left(\left\lceil \frac{w_k}{10} \right\rceil \times \left\lceil \frac{M}{36} \right\rceil\right)} \approx \frac{kc}{100} \times \frac{w_k + w_v}{w_k}.$$

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

$$\frac{BRAM_{dmhc\_match}}{BRAM_{fully\_assoc\_match}} \approx \frac{kc}{50}. \tag{10}$$

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. Increasing the dMHC Advantage

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 composed of a w_k-bit-wide *key* field and a w_v-bit-wide *value* field. 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 Figure 2. This could directly translate into a very wide G table whenever the key is very wide and/or the data value is very wide. On top of this, our architecture has to store these fields k times for k hash functions. For the rest of the article, we refer to this design as the Flat dMHC design.

In the ideal case, we would like to only keep a single copy of all the key–value pairs (instead of k copies). We can modify the Flat dMHC design 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 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, as well as yield the associated value. The resulting dMHC architecture is shown in Figure 4. As we can see in the figure, this new design results in a two-BRAM-cycle access; hence, we call it the two-level dMHC. The two-cycle access with a level of indirection is similar to the perfect hash design in Czech et al. [1992].

For a dMHC with M entries, the addresses are only $\log_2(M)$ bits wide. 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 two-level dMHC for any (k, c). This can result in a significant reduction in BRAM consumption for the G tables as the two-level dMHC G table widths are independent of the width of the key–value pairs.

Modifying Equation (8) for the two-level dMHC design, we get

$$BRAM_{dmhc\_match\_2level} = k \times \left\lceil \frac{\log_2(M)}{36} \right\rceil \times \left\lceil \frac{cM}{1024} \right\rceil. \tag{11}$$

[1]LUT usage is discussed in Online Appendix C.

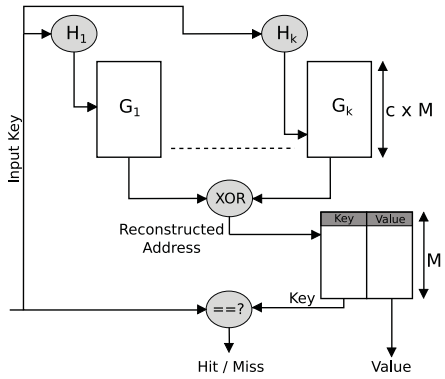


Fig. 4. Two-level dMHC(k,c).

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

$$\frac{BRAM_{dmhc\_match\_2level}}{BRAM_{fully\_assoc\_match}} = \frac{k \times \left\lceil \frac{\log_2(M)}{36} \right\rceil \times \left\lceil \frac{cM}{1024} \right\rceil}{\left(\left\lceil \frac{w_k}{10} \right\rceil \times \left\lceil \frac{M}{36} \right\rceil \right)} \approx \frac{kc}{100} \times \frac{\log_2(M)}{w_k}.$$

In comparison to the Flat dMHC design, the two-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 two-level* dMHC *with* $w_k = 64$ *bits, a* dMHC*(4,2) with 1,024 entries would consume* $\frac{1}{80}^{th}$ *of the BRAMs consumed by the fully associative memory*—roughly 14× less than the flat dMHC design.

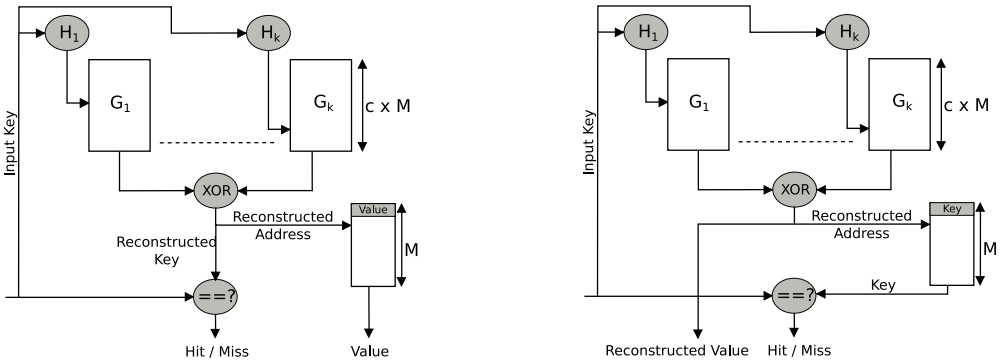
3.7. Performance-Area Hybrid dMHC Variants

The Flat dMHC (Figure 2) gives us a single BRAM cycle latency but consumes a large number of BRAMs. The two-level dMHC (Figure 4) consumes significantly fewer BRAMs but results in a two-BRAM-cycle latency. Even for the latency-sensitive cases, there could be two cases:

—Where we need to know if the key–value is present in the memory as soon as possible
—Where we need the data value quickly and we can confirm the presence in the memory later

It is possible to modify our two-level dMHC to achieve both these cases. If we simply add the key fields back into the G tables, this allows us to reconstruct the key in the first BRAM cycle and signal the rest of the system if it is found in the memory (with a false-positive rate as discussed in Section 3.4) or not in the first BRAM cycle. At the same time, we fetch the address of the data value, which can then be read in the second BRAM cycle. This gives us the Fast-Match dMHC as shown in Figure 5(a), with the match result in the first cycle and value in the second cycle.

Alternately, if we add the data value fields in the G tables, then we can simply reconstruct the value in the first BRAM cycle and match the key in the second BRAM cycle, giving us the Fast-Value dMHC as shown in Figure 5(b). This variant is suitable for use in a processor pipeline that speculates that a cache hit occurs and later squashes and reissues the instruction in the uncommon case of a miss. Table II captures the BRAM cost and latencies for all four dMHC variants where latency is the time taken to get the associated value after the key is presented.



(a) Fast-Match dMHC (b) Fast-Value dMHC

Fig. 5. Performance-area hybrid dMHC variants.

Table II. Comparison of Various dMHC Variants

dMHC Design	G Table BRAMs	Latency		False Positives?
		Match	Value	
Flat	$O((w_k + w_v + \log_2 M) \times M)$	1 cycle	1 cycle	Yes
2-level	$O(M \times \log_2 M)$	2 cycles	2 cycles	No
Fast-Match	$O((w_k + \log_2 M) \times M)$	1 cycle	2 cycles	Yes
Fast-Value	$O((w_v + \log_2 M) \times M)$	2 cycles	1 cycle	No

3.8. Minimum Area to Achieve a Target Miss Rate

Our primary goal is to achieve a near-associative memory performance without paying a high BRAM cost for a fully associative memory. In this section, we show we can configure a dMHC instance to achieve an arbitrarily low conflict probability with the least number of BRAMs possible. Let us assume a dMHC of M entries with w_k-bit keys and w_v-bit data values. Ideally, there may be multiple ways to achieve a particular collision rate since there could be multiple (k, c)-pairs that achieve the same collision probability (see Equation (5)). Thus, it should be possible to choose the BRAM-optimal dMHC configuration to achieve a given collision probability for a given memory capacity.

Since the parameter c should be a power of 2, let $c = 2^g$. From Equation (5), we have

$$P_{conflict} \approx \frac{1}{c^k} = 2^{-kg}. \tag{12}$$

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

$$2^{-kg} = 2^{-n} \text{ or, } kg = n. \tag{13}$$

For example, $n = 16$ gives a collision probability of 1 in 65,536. With $n = 16$, we have the options of implementing a dMHC with $(g = 1, k = 16)$ to $(g = 16, k = 1)$. To minimize the BRAM consumption, we can make this decision based on the number of BRAMs consumed for each of the previous configurations. For this, we only consider the number of BRAMs consumed by the match unit (i.e., G tables). We start with Equation (8). Since the G table width $(w_k + w_v$ in the flat case in Equation (8) or $\log_2(M)$ in the two-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.

$$BRAM_{dmhc\_match}(k, c) = \alpha \times k \times \left\lceil \frac{cM}{1024} \right\rceil \tag{14}$$

Now, $k = \frac{n}{g} = \frac{n}{\log_2(c)}$. Letting $\beta = \frac{\alpha}{1024}$,

$$BRAM_{dmhc\_match}(c) \approx \beta \times \frac{ncM}{\log_2(c)}. \quad (15)$$

Taking the derivative of Equation (15) with respect to 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 Section 5, we experimentally show that $c = 2$ is sufficient to achieve a near-associative performance.

3.9. Configuring dMHC to Achieve a Target BRAM Consumption

The previous section showed how we could configure a dMHC to achieve a target conflict rate (guaranteed statistically) while minimizing the number of BRAMs. In this section, we show how we can also configure a dMHC to achieve a target BRAM consumption while minimizing the conflict rate. Let us assume that we have a design target of B BRAMs. Again, since the G table width will depend on the dMHC variant used and is independent of the parameters k and c, we will assume it to be α. Therefore, total BRAMs for the match portion is

$$B = k \times \alpha \times \left\lceil \frac{cM}{1024} \right\rceil \approx k \times \beta \times c, \quad (16)$$

assuming $\beta \approx \frac{\alpha \times M}{1024}$. At the same time, we want to minimize conflict-rate ($\frac{1}{c^k}$). Let

$$conflict\_rate(k, c) = \frac{1}{c^k}. \quad (17)$$

Replacing $k = \frac{B}{\beta \times c}$,

$$conflict\_rate(k, c) = \frac{1}{c^{\frac{B}{\beta \times c}}}. \quad (18)$$

The previous expression is again minimized for $c = e$, giving us $k = \lceil \frac{B}{\beta \times e} \rceil$. Using these expressions, we can instantiate a dMHC(k,c) that meets a given BRAM target and achieves the lowest statistically guaranteed miss rate. This is consistent with the result in the previous section that it is ideal to set $c = 2$ or $c = 4$.

4. dMHC MEMORY MANAGEMENT

So far we have discussed the hardware organization of the dMHC architecture and how we can tune its configuration to achieve an arbitrarily low conflict rate. To manage an M-entry dMHC dynamically, holding at most M key–value pairs at a time, we will need to delete and insert values in the memory from time to time. In this section, we present our novel memory management algorithm and show how we can further reduce the number of conflicts. The remainder of this section describes the details of the state and operations needed to implement our management algorithm.

4.1. Table Composition

We store the complete key–value pairs in their original form in a memory called the M table, and we refer to each entry in there as an M slot. We refer to each entry in a G table as a G slot. The actual composition of a G slot will vary with the dMHC variant. Figure 6 shows the composition of an M slot and a G slot for the Flat dMHC. For each G slot, we store the number of M slots using that particular G slot in a field called the *degree*. The G slot fields *MRU*, *addr,* and *degree* are common to all the dMHC variants. A 2-Level dMHC only needs the *MRU, addr,* and *degree* fields. For the Fast-Match dMHC, the G slot

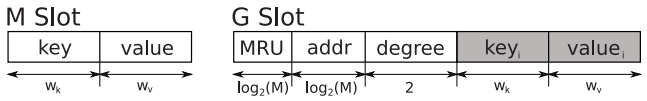


Fig. 6. G and M slots for a dMHC with capacity M, w_k-bit keys, and w_v-bit values.

also includes the key_i field, and a Fast-Value dMHC G slot includes the $value_i$ field. 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, as 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 conflict miss. On the other hand, associative memories have 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 (even a fully associative memory needs to evict entries in case of a capacity miss), requiring some cleanup of the state (Section 4.3).

There exist many eviction policies such as LRU and LFU. For our experiments in this work, we used the FIFO policy that evicts the least recently inserted entry. The FIFO policy may not be as effective as LRU or LFU in general, but it requires much less state to be maintained; LRU requires that we keep the age of each entry, whereas FIFO can be implemented simply as a single global counter.

4.3. Cleanup on Eviction

As explained in Section 3, each key–value pair is stored by assigning suitable values to the k G slots used by the key. Moreover, the collision probability in Equation (5) 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 each G table. Assuming uniformly distributed hash functions, the used G slots are uniformly distributed. When we are evicting a key–value pair, if we do not clean up 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 collision probability computed in Equation (5). 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 conflict probability as given by Equation (5). 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 [Fan et al. 2000]. Therefore, we need only reset those G slots that have a degree *1*, 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 continuous cleanup, the probability of all G slots being used by two or more key–value pairs is less than $(2c^2)^{-k}$, which is 0.02% for a dMHC(4,2) (see Online Appendix A). In order to corroborate this analytical result, we simulated a dMHC(4,2) for a set of SPEC CPU2006 benchmark programs and recorded the degree of G slots for each eviction. For $k = 4$, $c = 2$, $M = 1{,}024$, 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 (we use 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 begin to 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); see Online Appendix A), the G slots used 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-0 G slot that 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 k of the G slots can now reproduce the original key–value pair for the Flat dMHC design or the location in the memory for the two-level dMHC design. This requires the same XOR calculations as shown in Equation (3). At the same time, we increment the degree of all the G slots used by the new key–value pair. However, in the uncommon event where all the G slots are in use, we have to perform more work in order to resolve the conflict.

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; see Online Appendix A), all the G slots used 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 reassign 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, reassigning 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 as we did not really evict them from the memory. Nevertheless, in order to be able to insert a new key–value pair, we must reassign values in at least one G slot, and prudence tells us that we should only reassign values in a single 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 $1 - (2c^2)^{-k}$ (0.9986 for dMHC(4,2); see Online Appendix B). Once we are able to locate a G slot that has a degree of 1, we can reassign 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 reassigning 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 *reinsert* the victimized key–value pair by modifying the fields in another of its remaining $(k - 1)$ G slots. Continuing the idea of Equation (5), with a probability of $1 - c^{1-k}$, we can *reinsert* this entry by modifying a G slot that is being used only by this key–value pair. Here the collision probability is c^{1-k} rather than c^{-k} because we know it will conflict with the newly inserted entry that caused this key–value pair to be victimized in the first place. However, with a very low probability (less than $(2c^2)^{-k}$), we create another conflict (when the G slot chosen to reinsert the

victimized key–value pair has a degree greater than 1). In that case, we continue removing and reinserting 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 reinsert the key–value pair victimized in the previous hop. This is equivalent to moving a hash entry to accommodate an insertion (*c.f.* Kirsch and Mitzenmacher [2010]).

In order to be able to evict and reinsert 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 two-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 pair in the most common case, and even in the uncommon case, reinserting only the most recently accessed value is in favor of temporal locality.

When we do victimize more than one key–value pair (less than 0.14% of the time for `dMHC`(4,2)), two things go bad: (1) since we only reinsert 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 (2) the G slots storing information for these key–value pairs are not cleaned up as explained in Section 4.3, affecting the conflict miss probability. However, since the FIFO 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 clean up their G slots.

4.6. Lowest Degree Victim with N-Hop Repair

Generalizing the previous strategy, 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 that would victimize the least number of M slots. Once an entry is victimized, we can then try to repair it as explained in the previous section; we call it the N-hop repair strategy since we follow chains up to length N to resolve conflicts. Algorithm 1 shows the complete algorithm for Lowest Degree Victim with N-hop Repair (or `LDVN` in short) for `dMHC` memory management.

4.7. Characterizing N-Hop Repair Algorithm

In this section, we characterize the `LDVN` management algorithm in terms of the state that it needs to maintain as well as its latency. For simplicity, let us first analyze the state needed for the `LDV0` (Lowest Degree Victim with 0 hops) policy and then build on that. Let us assume the case where the memory is at capacity. In that case, first, we need to evict an entry as per the FIFO policy. As mentioned in Section 4.3, we perform cleanup of the state used by the evicted key–value pair. This requires the following steps:

(1) Compute the k hashes of the evicted key (Algorithm 1, line 4), which can be done using the logic for the existing hash functions.
(2) Read the k G slots that are storing that key–value pair.
(3) Check the degree for each G slot; if the degree is 1, we reset that slot with 0s; otherwise, we decrement the degree by one (Algorithm 1, lines 40–46).
(4) Update the G slots as per the previous step by writing to the corresponding G tables.

ALGORITHM 1: Pseudocode for Lowest Degree Victim with N-hop Repair Policy for dMHC (4,2)

```
1  Function LDVN(K, V, N)
2  // K is the input key, V is the value, N is the num. of hops
3  new_m ← m_slot_counter  //global FIFO counter
4  h_i ← H_i(M[new_m].key) for 1 ≤ i ≤ k
5  cleanup(G_i[h_i]) for 1 ≤ i ≤ k
6  h_i ← H_i(K) for 1 ≤ i ≤ k
7  if there is an unused G_i[h_i] then
8  |  use G_i[h_i] to store {K, V} at M[new_m]
9  else
10 |  choose i s.t. deg(G_i[h_i]) =
11 |           min_deg(G_1[h_1]...G_k[h_k])
12 |  reassign G_i[h_i].{key, val, addr} to store {K, V} at M[new_m]
13 |  victim_m ← G_i[h_i].mru
14 |  LDV(victim_m, i, N);
15 end
16 G_i[h_i].degree + + for 1 ≤ i ≤ k
17 G_i[h_i].mru ← new_m for 1 ≤ i ≤ k
18 m_slot_counter + + // FIFO replacement of M slots
19 return dMHC_miss
20
21 Function LDV(m_slot, j, N)
22 if N = 0 then
23 |  return 0  //no more hops
24 else
25 |  h_i ← H_i(M[m_slot].key) for 1 ≤ i ≤ k
26 |  choose i ≠ j s.t. deg(G_i[h_i]) =
27 |           min_deg(G_1[h_1]...G_k[h_k])
28 |  reassign G_i[h_i].{key, val, addr} to store
29 |           M[m_slot].key at M[m_slot]  //update only the chosen G slot
30 |  if G_i[h_i].degree = 1 then
31 |  |  return 0  //no more hops
32 |  else
33 |  |  victim_m ← G_i[h_i].mru
34 |  |  G_i[h_i].mru ← m_slot
35 |  |  return LDV(victim_m, i, N − 1)
36 |  end
37 end
38 EndFunction
39
40 Function cleanup(g_slot)
41 if g_slot.degree=1 then
42 |  reset g_slot to 0
43 else
44 |  if g_slot.degree≠ 0 then
45 |  |  g_slot.degree ← g_slot.degree - 1
46 |  end
47 end
48 EndFunction
```

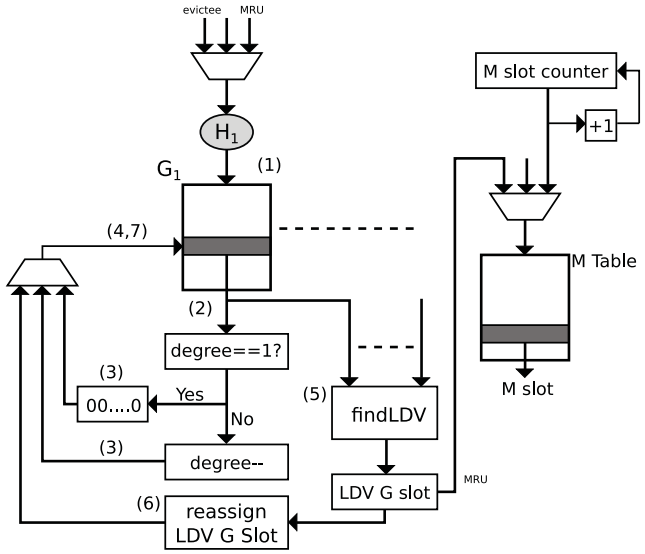


Fig. 7. Spatial implementation of the LDVN policy.

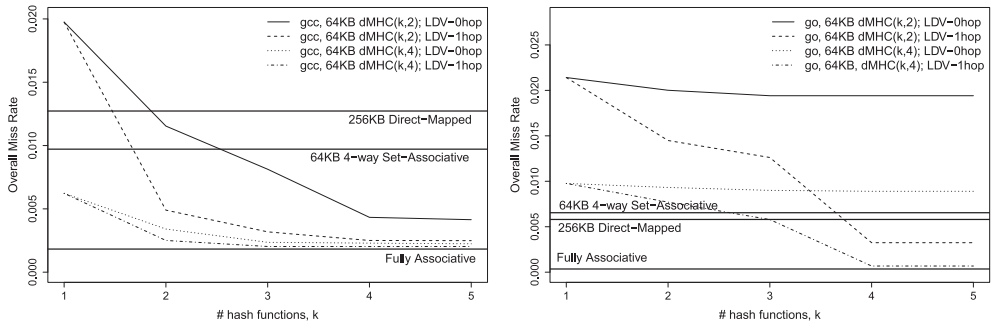
This requires k registers, each as wide as a G slot, k 2-bit compare-to-one circuits and k 2-bit decrement-by-one circuits. Figure 7 shows how this can be implemented spatially. As can be seen in the figure (steps marked with corresponding numbers), we can perform these operations in four cycles.

The next step is to insert the new key–value pair; the worst case is when we have to victimize a G slot. In that case:

(5) First, we have to determine the G slot with the lowest degree (line 10). This can be implemented spatially as a k 2-bit input minimum circuit with $3(k-1)$ 6-LUTs.
(6) Once the victim G slot is determined, we need to reassign the value to that G slot in order to insert the new key–value pair (line 12). This requires at most $(w_k + w_v) \times \lceil \frac{k}{6} \rceil$ 6-LUTs (for the Flat dMHC design). At the same time, we have to increment the degrees of all the G slots used to store the information for the new key–value pair, requiring k 2-bit increment-by-one circuits and a write of the current FIFO counter as the MRU for these G slots (Algorithm 1, lines 16–17).
(7) Finally, we write the updated G slots back to the corresponding tables.

These operations consume another three cycles, assuming the LDV is found in a single cycle, bringing the latency of the LDV0 policy to seven cycles.

Now, for the LDV1 policy, we need to perform all the operations for LDV0, with the addition of repairing the MRU for the victim G slot (in case there is one, lines 13–14). The repair procedure is same as the LDV0 itself, wherein now we are trying to insert the key–value pair that we just falsely evicted. Therefore, LDV1 adds another seven cycles. Similarly, each additional hop adds seven cycles to the latency and no additional hardware, making the latency of the LDVN policy $7(N+1)$ cycles. Assuming that the next level memory in the memory hierarchy is external to the FPGA chip (say, an external DRAM), it might take hundreds of cycles to get the new key–value pair. This means that we could take up to 15 hops without adding to the latency of the miss-service algorithm. Another thing to note is that we must avoid victimizing the same G slot twice in a chain of repairs; otherwise, it could lead to nonterminating



(a) gcc (b) go

Fig. 8. Miss rates for a 64KB dMHC.

cycles. Furthermore, we also need to add logic to keep track of the number of hops we have taken and to abort the LDVN procedure once the number of hops taken reaches N, or we reach a point where no further repairs can be performed. For a dMHC(4,2) with a 64-bit key width and a 64-bit value, we are able to implement the LDV1 with 189 6-LUTs, achieving a frequency of operation around 400MHz.

5. PERFORMANCE COMPARISON

In this section, we present FPGA implementation details for the dMHC designs as well as simulation results for a set of memory-intensive benchmarks.

5.1. Hardware Implementation

We implemented the proposed dMHC architecture in Bluespec SystemVerilog (BSV) [Bluespec, Inc. 2012] hardware description language. Our tool[2] can generate a parameterized dMHC instance to target a particular conflict rate or a BRAM budget. Using the BSV compiler, the tool generates Verilog HDL code that can then be synthesized using Xilinx ISE tools. We also implemented the LDV0 and LDV1 policies for memory management directly in BSV as low-level control FSMs. In order to reduce the miss-service latency in the memory controller, we have implemented both policies as spatially as possible (Figure 7).

5.2. Case Study I: 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 (Section 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 SPEC CPU2006 Benchmark Suite [Henning 2006] using traces from a 64-bit x86-simulator [Battle et al. 2012] and simulating each benchmark for 100M cycles. Memory reference counts for the address traces used in the present work are highlighted in Table III (column 1).

In order to perform a direct comparison, we also simulated a fully associative memory and several set-associative caches for the same benchmarks. Figure 8 shows how the

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

Table III. Fully Associative–to–dMHC BRAM Ratio for a 64KB L1 D-Cache for Eight SPEC CPU2006 Benchmarks

Benchmark (Mem instrs)	dMHC config (k,c)	Conflict Ratio Theoretical (%)	Conflict Ratio Observed (%)	Flat BRAM Ratio	2-level BRAM Ratio	Fast-Match BRAM Ratio	Fast-Value BRAM Ratio
art (19.9M)	(4,2)	6.25	2.6	3.6	10.8	6.5	4.6
gcc (31.4M)	(4,2)	6.25	3.2	3.6	10.8	6.5	4.6
go (35.5M)	(4,2)	6.25	3.8	3.6	10.8	6.5	4.6
hmmer (41.9M)	(3,2)	12.5	1.4	4.6	13.0	8.1	5.9
libq (30.2M)	(2,2)	25.0	0.04	6.5	16.2	10.8	8.1
mcf (32.2M)	(4,2)	6.25	2.8	3.6	10.8	6.5	4.6
sjeng (26.9M)	(4,2)	6.25	4.9	3.6	10.8	6.5	4.6
sphinx3 (33.1M)	(3,2)	12.5	4.6	4.6	13.0	8.1	5.9

overall miss rate varies for our architecture with respect to the parameters k and c for the benchmarks gcc and go for a 64KB L1 data cache with a cache line size of 64 bytes. The miss rate is the same for all dMHC variants. The figure also shows the miss rate achieved with a fully associative memory of the same capacity as the dMHC, a direct-mapped cache with four times the capacity, and a four-way set-associative cache of the 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 the same capacity. One can also observe that the LDV1 policy clearly outperforms the LDV0 policy for the same dMHC configurations. In Section 5.3, we compare the BRAM consumption for these caches.

5.3. Case Study II: L1 Data Cache BRAMs

Table III shows the BRAM usage ratio for eight SPEC CPU2006 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 defined arbitrarily as when less than 5% of misses are due to conflicts. In each row, we indicate the conflict ratio as suggested by Equation (6) and the actual conflict ratio observed by simulating the benchmarks. Equation (6) gives us the raw conflict probability defined by the dMHC configuration parameters. However, with our LDV1 policy, we are able to reduce the number of conflicts below the raw value by performing repairs on accidently evicted values. For each chosen configuration, we also report the fully associative–to–dMHC BRAM usage ratio for all the four variants. From the data in Table III, we observe that a dMHC(4,2) configuration with LDV1 policy is able to achieve target conflict ratios for all the benchmarks with fewer BRAMs compared to a fully associative memory.

Results from Table III show that our architecture is able to achieve a near-associative performance for a dMHC(4,2) configuration. We further ran simulations with various cache organizations (direct mapped, set associative, fully associative, and dMHC) varying the capacity from 1KB to the point where we saturate all the BRAMs available on a Virtex 6 (xc6vlx240t-2 device) FPGA, and for each cache size, we record the miss rate achieved and the number of BRAMs consumed.

Figure 9 shows how the miss rate falls when we increased the capacity of these caches in terms of BRAMs for the benchmark gcc. For any type of cache, increasing the number of BRAMs increases capacity, and therefore reduces capacity misses. From Figure 9, 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.

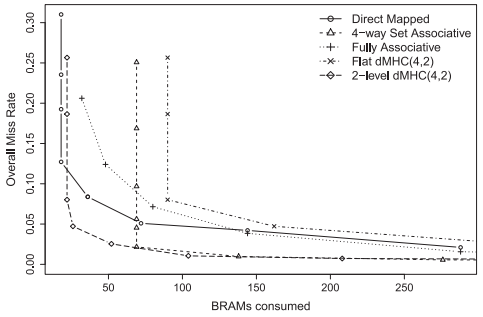


Fig. 9. BRAMs versus miss rates for `gcc`.

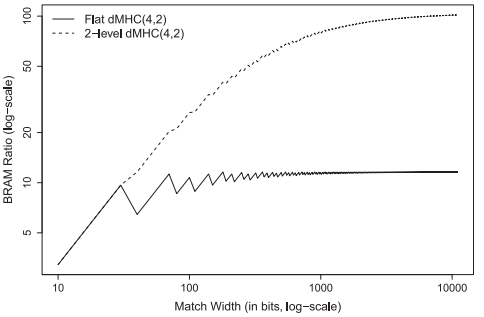


Fig. 10. Fully associative memory–to–dMHC(4,2) BRAM usage ratio.

5.4. `dMHC` as a Choice of Cache for Extreme Cases

Using the `dMHC` designs as a small L1 data cache, we showed that we can achieve a high memory performance with significantly fewer BRAMs. Furthermore, the `dMHC` architecture can achieve even higher BRAM savings as the match width is increased. For example, Dhawan et al. [2012] mention a need to map 231-bit keys to 201-bit values. Figure 10 shows the BRAM savings for the `Flat` as well as the `2-level dMHC` designs over the Coregen-style fully associative memory, all of depth 1,024 entries (this is only for the key-match portion). For the `Flat` variant, the saving ratio saturates at about 11×, whereas the `2-level` variant is able to save up to 100× over the fully associative memory.

5.5. Limit-Study Experiment for the `LDVN` Policy

In this section, we perform a limit study on the `LDVN` policy using the address-trace simulations described earlier. We use a `dMHC(4,2)` and `dMHC(4,4)` as a 64KB L1 data cache for the same set of SPEC CPU2006 benchmarks and simulate these `dMHC` instances with an LDV∞ policy (i.e., an unbounded number of hops). For each benchmark, we record the maximum number of hops taken to resolve all the conflicts or if the algorithm reaches a point where no more conflicts can be resolved without disrupting already resolved conflicts. Figure 11 shows the maximum number of hops taken by each benchmark—all the benchmarks except `hmmer`, `mcf`, and `sjeng` saturate at the same number of hops for both $c = 2$ and $c = 4$. For these three cases, $c = 2$ takes one extra hop in the worst case. The figure also shows the average number of hops taken for each of the benchmarks. The average number of hops taken across all the benchmarks is less than 1, suggesting that an `LDV1` policy should be enough to achieve a near-associative performance. With $c = 2$, we were able to resolve all the $k-collisions$ for all of the benchmarks except for `hmmer`. Moving from `LDV1` to `LDV2` policy provides an improvement of only 0.05% on average.

5.6. `dMHC` Timing

Another disadvantage of the Coregen-style fully associative memory is the low frequency of operation. Reviewing Figure 1(b) shows that 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, 1,024). 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 1,024-entry `dMHC` and fully associative

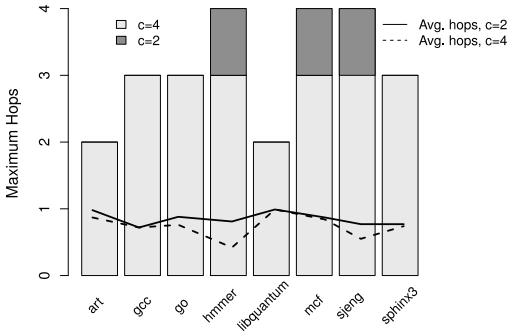


Fig. 11. Maximum and average number of hops for a `dMHC(4,c)`.

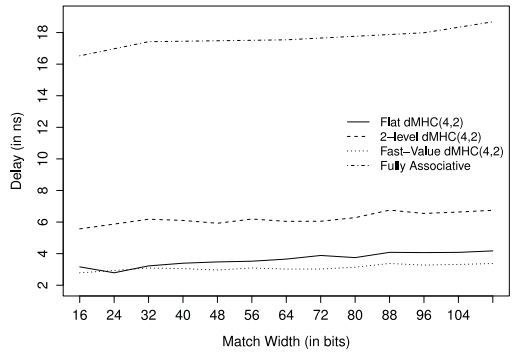


Fig. 12. `dMHC` versus fully associative delays for a 1,024-entry memory holding 64-bit values.

memory designs with 64b values and varying key widths from 16-bit to 120-bit. These designs were then placed and routed using Xilinx ISE 13.2 for a Virtex 6 (xc6vlx240t-2) device. Figure 12 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 in an unpipelined design. Here we show access latency, whereas Dhawan and DeHon [2013] only showed the pipeline cycle time. Along with different BRAM footprints, the `Flat` and the two-level `dMHC` designs have slightly different critical paths. Apart from that, the fully associative memory and two-level `dMHC` each require two BRAM cycles to fetch the data value in the most common case; pipelined, these both can launch one memory lookup per BRAM cycle. Using the two-level `Fast-Value 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 [1993] 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 two-way skewed-associative cache has a miss rate close to a regular four-way set-associative cache with the hardware complexity of a two-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 [Kirsch and Mitzenmacher 2010]. 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 [Sanchez and Kozyrakis 2010]. 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` as well as the `Fast-Value dMHC`s produce the candidate result after a single memory lookup.

Bloomier filters [Chazelle et al. 2004] extend Bloom filters by giving the exact pattern that matched along with the set membership. These have been effectively used in applications such as accelerating virus detection using FPGA hardware [Ho and

Lemieux 2008]; 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 [Song et al. 2005]. 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 in 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 near-associative memory performance. Furthermore, we have shown how it can be parameterized in terms of capacity, k, c, and design variants. We also showed that the proposed architecture can be easily tuned in order to engineer the BRAM usage, conflict rate, and/or access latency to the memory. We showed that the 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 (Section 5.3). Furthermore, we've shown that the dMHC implementations have lower access latency (Figure 12). The dMHC should be in any FPGA application or reconfigurable computing designer's arsenal of building blocks.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library.

ACKNOWLEDGMENTS

This material is based on 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. The authors would like to thank Xilinx Inc. for generous donations of ISE tools and ML605 development boards and Bluespec Inc. for donation of Bluespec tools. The authors would also like to acknowledge Rafi Rubin for his critical remarks on the text.

REFERENCES

Yossi Azar, Andrei Z. Border, Anna R. Karlin, and Eli Upfal. 1994. Balanced allocation. In *Proceedings of the ACM Symposium on Theory of Computing*. 593–602.

Steven Battle, Andrew D. Hilton, Mark Hempstead, and Amir Roth. 2012. Flexible register management using reference counting. In *Proceedings of the International Symposium on High-Performance Computer Architecture*. IEEE, 273–284. DOI:http://dx.doi.org/10.1109/HPCA.2012.6169033

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

Bluespec, Inc. 2012. Bluespec SystemVerilog 2012.01.A. Retrieved from http://www.bluespec.com.

Suzanne Bunton and Gaetano Borriello. 1992. Practical dictionary management for hardware data compression. *Commun. ACM* 35, 1 (1992), 95–104. DOI:http://dx.doi.org/10.1145/129617.129622.

Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. 2004. The Bloomier filter: An efficient data structure for static support lookup tables. In *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA'04)*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 30–39.

Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. 1992. An optimal algorithm for generating minimal perfect hash functions. *Inform. Process. Lett.* 43, 5 (1992), 257–264.

Udit Dhawan and André DeHon. 2013. Area-efficient near-associative memories on FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. 191–200.

Udit Dhawan, Albert Kwon, Edin Kadric, Cătălin Hriţcu, Benjamin C. Pierce, Jonathan M. Smith, Gregory Malecha, Greg Morrisett, Thomas F. Knight, Jr., Andrew Sutherland, Tom Hawkins, Amanda Zyxnfryx, David Wittenberg, Peter Trei, Sumit Ray, Greg Sullivan, and André DeHon. 2012. Hardware support for safety interlocks and introspection. In *Proceedings of the SASO Workshop on Adaptive Host and Network Security*. http://ic.ese.upenn.edu/pdf/interlocks_ahns2012.pdf.

Li Fan, Pai Cao, Jussara Almeida, and Andrei Z. Border. 2000. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Networking* 8, 3 (2000), 281–293.

John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News* 34, 4 (September 2006), 1–17. DOI:http://dx.doi.org/10.1145/1186736.1186737

J. Ho and G. Lemieux. 2008. PERG: A scalable FPGA-based pattern-matching engine with consolidated Bloomier filters. In *Proceedings of the International Conference on Field-Programmable Technology*. 73–80. DOI:http://dx.doi.org/10.1109/FPT.2008.4762368

Adam Kirsch and Michael Mitzenmacher. 2010. The power of one move: Hashing schemes for hardware. *IEEE/ACM Trans. Networking* 18, 6 (2010), 1752–1765.

Charles Eric LaForest and Gregory Steffan. 2012. Octavo: An FPGA-centric processor family. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. 97–106.

Shih-Lien L. Lu, Peter Yiannacouras, Taeweon Suh, Rolf Kassa, and Michael Konow. 2008. A desktop computer with a reconfigurable Pentium. *ACM Transactions on Reconfigurable Technology and Systems* 1, 1 (March 2008).

Michael Mitzenmacher. 1999. Studying balanced allocation with differential equations. *Combin. Probab. Comput.* 8, 5 (1999), 473–482.

Jad Naous, David Erickson, G. Adam Covington, Guido Appenzeller, and Nick McKeown. 2008. Implementing an OpenFlow switch on the NetFPGA platform. In *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. 1–9. DOI:http://dx.doi.org/10.1145/1477942.1477944

Daniel Sanchez and Christos Kozyrakis. 2010. The ZCache: Decoupling ways and associativity. In *Proceedings of the International Symposium on Microarchitecture*. 196–207.

André Seznec. 1993. A case for two-way skewed-associative caches. In *Proceedings of the International Symposium on Computer Architecture*. 169–178.

André Seznec and François Bodin. 1993. Skewed-associative caches. In *Parallel Architectures and Languages Europe*. 304–316. DOI:http://dx.doi.org/10.1007/3-540-56891-3_24

Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. 2005. 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*. 181–192. DOI:http://dx.doi.org/10.1145/1080091.1080114

John Wawrzynek, David Patterson, Mark Oskin, Shih-Lien Lu, Christoforos Kozyrakis, James C. Hoe, Derek Chiou, and Krste Asanović. 2007. RAMP: Research accelerator for multiple processors. *IEEE Micro* 27, 2 (2007), 46–57.

Sewook Wee, Jared Casper, Njuguna Njoroge, Yuriy Tesylar, Daxia Ge, Christos Kozyrakis, and Kunle Olukotun. 2007. A practical FPGA based framework for novel CMP research. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays*. 116–125.

Rondald Wunderlich and James C. Hoe. 2004. In-system FPGA prototyping of an itanium microarchitecture. In *Proceedings of the International Conference on Computer Design*. 288–294.

Xilinx, Inc. 2011a. *Parameterizable Content-Addressable Memory*. Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124. XAPP 1151 http://www.xilinx.com/support/documentation/application_notes/xapp1151_Param_CAM.pdf.

Xilinx, Inc. 2011b. *Virtex-6 FPGA Data Sheet: DC and Switching Characteristics*. Xilinx, Inc., 2100 Logic Drive, San Jose, CA 95124.

Peter Yiannacouras and Jonathan Rose. 2003. A parameterized automatic cache generator for FPGAs. In *Proceedings of the International Conference on Field-Programmable Technology*. 324–327.

Peter Yiannacouras, J. Gregory Steffan, and Jonathan Rose. 2007. Exploration and customization of FPGA-based soft processors. *IEEE Transactions on Computer-Aided Design* 26, 2 (2007), 266–277.

Received June 2013; revised October 2013; accepted January 2014