Hyperdimensional Hashing: A Robust and Efficient Dynamic Hash Table

Mike Heddes, Igor Nunes, Tony Givargis, Alexandru Nicolau and Alex Veidenbaum
Department of Computer Science, University of California, Irvine
Irvine, California, United States of America
{mheddes,igord,givargis,nicolau,alexv}@uci.edu

Abstract

Most cloud services and distributed applications rely on hashing algorithms that allow dynamic scaling of a robust and efficient hash table. Examples include AWS, Google Cloud and BitTorrent. Consistent and rendezvous hashing are algorithms that minimize key remapping as the hash table resizes. While memory errors in large-scale cloud deployments are common, neither algorithm offers both efficiency and robustness. Hyperdimensional Computing is an emerging computational model that has inherent efficiency, robustness and is well suited for vector or hardware acceleration. We propose Hyperdimensional (HD) hashing and show that it has the efficiency to be deployed in large systems. Moreover, a realistic level of memory errors causes more than 20% mismatches for consistent hashing while HD hashing remains unaffected.

CCS Concepts: • **Networks → Cloud computing**; • **Computer systems organization → Reliability**; • **Computing methodologies → Massively parallel algorithms**.

Keywords: hyperdimensional computing, brain-inspired computing, consistent hashing, rendezvous hashing, distributed hash tables, cloud computing, load balancing, web caching.

ACM Reference Format:
Mike Heddes, Igor Nunes, Tony Givargis, Alexandru Nicolau and Alex Veidenbaum. 2022. Hyperdimensional Hashing: A Robust and Efficient Dynamic Hash Table. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC) (DAC '22), July 10–14, 2022, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3489517.3530553

1 Introduction

An important problem in many cloud services and distributed network applications is the process of mapping requests to available resources. Example systems include: load balancing in cloud data centers, web caching, peer-to-peer (P2P) services, and distributed databases. Difficulty arises in such highly dynamic systems because resources join and leave the cluster at any time, due for example to cloud elasticity [1], server failures, or availability of peers in a P2P network. It is often desirable to distribute requests evenly among resources and to minimize the number of redistributed requests when a resource joins or leaves. A non-uniform mapping results in overloading of resources and critical failure points.

The simplest hash table solves the mapping problem using modular hashing. Despite having a great lookup time complexity of $O(1)$, a change in table size (number of available resources) requires virtually all requests to be redistributed due to the modulo operation (more details in Section 2). *Consistent hashing* [10] and *rendezvous hashing* [23] are alternative hashing algorithms that minimize redistribution when the hash table is resized. They prevent resource overloading at the cost of increased lookup time—$O(\log n)$ and $O(n)$ respectively.

However, we show that when considered in a dynamic environment subject to errors and failures (i.e., noise), the performance of consistent hashing and rendezvous hashing in minimizing the number of redistributed requests degrades. Noise can be introduced in many aspects of a system. We focus on memory errors which can for instance be caused by soft errors in the form of single event upsets (SEU), multi-cell upsets (MCUs) or hard errors [5, 21]. MCUs, or burst errors, occur during a single event and are becoming more common as the feature size decreases. For 22 nm technology MCUs are estimated to be 45% of all SEUs [6]. Moreover, analysis of memory failures in Google's data centers revealed that each year a third of the machines experiences a memory error [19]. More robust hashing alternatives make it possible for cloud providers to perform fewer memory swaps, reducing operation cost.

Hyperdimensional Computing (HDC) is an inherently robust emerging computational model developed by Kanerva [9] inspired by neuroscience. HDC tries to emulate brain-like computing by representing information using high-dimensional random vectors, called *hypervectors*. This representation shares qualities from biological neural systems such as robustness and efficiency. Representation and

DAC '22, July 10–14, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9142-9/22/07.
https://doi.org/10.1145/3489517.3530553

transformations of data in HDC are performed over hypervectors of fixed dimensions, allowing for massive parallelism.

Fueled by the demonstrated properties of HDC and the aforementioned limitations of current hashing algorithms, we propose *Hyperdimensional (HD) hashing*, a new HDC-based dynamic hashing algorithm. HD hashing scales similarly to consistent hashing while proving to be much more efficient than rendezvous hashing. HDC's highly parallelizable operations have been exploited in recent research, showing that special hardware can make HD hashing far superior in efficiency (more details in Sec. 2.3). Moreover, we show that our algorithm is significantly more robust against noise. With 512 servers and a 10-bit MCU, HD hashing is unaffected while rendezvous and consistent hashing mismatch 4% and 12% of requests, respectively. With MCUs becoming more common this poses a risk for critical failures.

Our second contribution is a novel HDC encoding for representing a circle in hyperdimensional space, we call these *circular-hypervectors*. They are a core component of HD hashing as they provide the mechanism for mapping requests to servers.

2 Background

2.1 Consistent Hashing

Consistent hashing is a common way of distributing requests among a changing population of servers [13, 22] (often times, the problem and the technique are referred to as *consistent hashing* indistinctly). The algorithm, which gave rise to Akamai [15], is used in many other real-world large scale applications such as Dynamo on Amazon Web Services [2] and Google Cloud Platform [3].

To describe consistent hashing, let $h(\cdot)$ denote a hash function that takes requests as inputs (in practice an IP address or unique identifier, for example) and $S = \{s_1, \ldots, s_n\}$ a set of servers. In modular hashing, a request r is simply assigned to s_i where $i = h(r) \bmod n$. Instead, consistent hashing maps both requests and servers uniformly to the unit interval $[0, 1]$, which is interpreted as a circular interval. Thereafter, each request is assigned to the first server that succeeds it in the circle in clockwise order. This assignment is usually done in $O(\log n)$ time using binary search.

2.2 Rendezvous Hashing

The basic idea of rendezvous hashing [23], also known as highest random weight (HRW) hashing, is very simple. Given a hash function $h(\cdot)$ that takes as input a server and a request, each request r is assigned to the server s_i where:

$$s_i = \arg\max_{s \in S} h(s, r)$$

Each assignment is therefore done in $O(n)$ time, since it is necessary to compute the hash of the request paired with each server in the system in order to compute the maximum value. In practice, Rendezvous hashing is used less often than consistent hashing, despite distributing the requests more uniformly, because of the increased time complexity.

2.3 Hyperdimensional Computing

From a comparative study of computing in animal brains and computer logic circuits [9], Hyperdimensional Computing (HDC) emerged as a robust and efficient alternative computation model. The central observation is that large circuits are fundamental to the brain's computation. HDC incorporates this notion by computing with 10,000-bit words (hypervectors), instead of 8-to-64-bit.

Such hyperspaces (short for hyperdimensional spaces) have properties that explain certain rich brain properties that are otherwise difficult to reproduce on computers. For example, hypervectors encode information holographically, meaning that each of the thousands of bits contains the same amount of information, ensuring inherent robustness [9, 25].

In addition to representation, the other crucial part of a computer system is information manipulation, or arithmetic. The arithmetic in HDC is based on well-defined operations between hypervectors, such as addition (bundling), multiplication (binding) and permutation. Another important function is information comparison, which in HDC usually means measuring the similarity between hypervectors using the inverse Hamming distance or the cosine similarity. All those operations are typically dimension-independent, providing an opportunity for massive parallelism [11, 17].

Computational efficiency is one of the core motivations aimed at since the conception of HDC and it is envisioned for and expected to reach full potential in specialized hardware [9]. In addition to the just mentioned parallelizability, optimizations such as in-memory processing promise to further increase the computational efficiency of HDC [7]. Schmuck et al. [18] apply a series of hardware techniques to optimize HDC, such as on-the-fly *rematerialization* of hypervectors and special memory architectures, to improve chip area and throughput at the same time. Particularly important to substantiate the claims we make in this paper about efficiency (see Section 3), they demonstrate an FPGA implementation that uses deep adder trees to perform inference in a single clock-cycle.

3 Hyperdimensional Hashing

HD hashing, illustrated in Figure 1, draws inspiration from consistent and rendezvous hashing, but seeks a solution that is both robust and efficient by translating the problem into a hyperdimensional computing task.

Let $S = \{s_1, \ldots, s_k\}$ be a set of k servers, $R = \{r_1, \ldots, r_\ell\}$ a set of ℓ requests and $C = \{\mathbf{c}_1, \ldots, \mathbf{c}_n\}$ a set of $n > k$ hypervectors. We also denote by $h(\cdot)$ a hash function that takes as input a server or request. The process of adding servers to the system in HD hashing is similar to consistent hashing, but instead of mapping them to a unit interval (see Sec. 2.1),

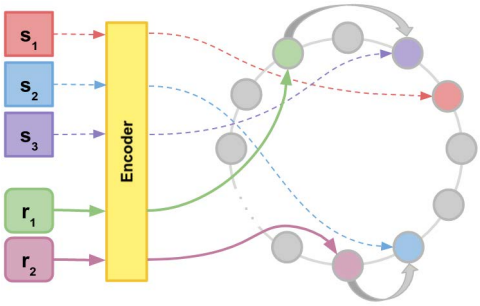

Figure 1. Illustration of the operation of HD hashing. In this example, after encoding each of the three servers and two requests to a circular-hypervector, r_1 is assigned to server s_3, which is the server whose hyperspace representation is closest to its. Likewise, r_2 is assigned to s_2. Note that, unlike consistent hashing, the direction of rotation does not matter.

HD hashing assigns (or "*encodes*" in HDC terminology) each server to a hypervector. To distribute requests among servers, HD hashing also encodes each request. Let us represent this encoding by the function $\text{Enc} : S \cup R \rightarrow C$. Then, HD hashing encodes every server and request as follows:

$$\text{Enc}(x) = C\big[h(x) \bmod n\big] \tag{1}$$

where x is either a server or a request and $C\,[h(x) \bmod n]$ denotes the hypervector at position $h(x) \bmod n$ in C.

With all servers and requests encoded to the hyperspace, each request r_i is mapped to server s_j, such that:

$$s_j = \arg\max_{s \in S} \delta\big(\text{Enc}(s), \text{Enc}(r_i)\big) \tag{2}$$

where δ is a given similarity metric between a pair of hypervectors such as inverse Hamming distance or the cosine similarity. The operation above is the one mentioned in Section 2.3, and it is called inference due to the first applications of HDC in learning tasks. This is exactly the operation that Schmuck et al. [18] show to be optimizable to the extreme of a single clock-cycle in special hardware. In other words, by using hardware accelerators for HDC each mapping in HD hashing could be executed in $O(1)$ time.

One remaining, but crucial, question is: how do we create the set of hypervectors C? Similar to consistent hashing, we map servers and requests onto a circle. We then map the request to the server that is assigned to the nearest node on the circle according to Eq. 2. To accomplish this, we introduce *circular-hypervectors* as a way of representing a circle in hyperspace such that the closer a node is on the circle the more similar its hypervector. More properties of circular-hypervectors and the process to create them are described in the next section.

4 Circular-Hypervectors

To understand circular-hypervectors we first describe random and level-hypervectors, both types are used to represent

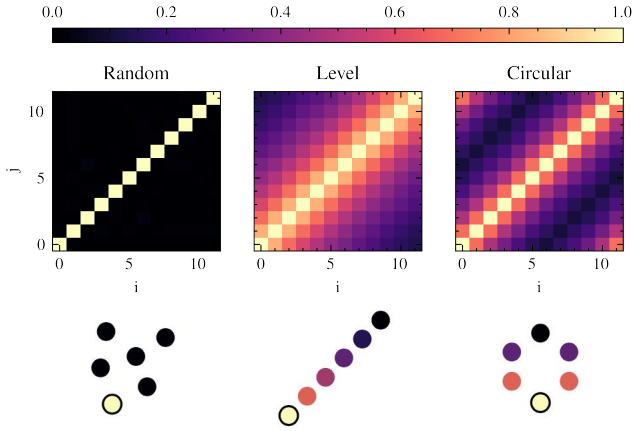

Figure 2. Pairwise cosine similarities between hypervectors i and j within different sets of 12 basis-hypervectors. An alternative visualization in which each hypervector is represented by a node is shown below. The colors indicate the similarity with the yellow reference node.

information in hyperspace, a process called *encoding*. Encoding strategies have already been proposed for various types of input data, such as images [12], time series [8] and text [14]. The process usually starts by generating a set of randomly sampled hypervectors that represent discrete atomic pieces of information (e.g. discretized amplitudes of a signal, values of a feature, symbols or identifiers). From these so-called basis-hypervectors more complex objects like the ones listed above can be encoded by combining and manipulating the basis-hypervectors using bundling, binding and permutation operations.

The basis-hypervectors can be correlated with each other depending on what they represent. For example, consider temperature. Clearly there is a stronger correlation between closer temperatures. On the other hand, for symbols such as letters, this correlation does not necessarily exist. Naturally, the most successful encoding techniques are able to translate these correlations into hyperspace. For this reason, categorical data (letters for example) are encoded with independently and uniformly sampled *random-hypervectors*, while scalar information (e.g. temperature) is represented using *level-hypervectors* [16].

Level-hypervectors are created by quantizing an interval to m levels and assigning a hypervector to each. The similarity between hypervectors is proportional to the distance between the intervals. This correlation is achieved by assigning a random d-dimensional hypervector to the first interval, and after this, subsequent intervals are obtained by flipping d/m random bits at each interval. As a result, the last hypervector is completely dissimilar to the first one.

Circular-hypervectors are an extension to level-hypervectors that eliminate the discontinuity in similarity between the last and first interval, as visualized in the similarity profiles

in Figure 2. By removing the discontinuity, the hypervectors become a set with circular correlation.

The procedure for generating circular-hypervectors, illustrated in Figure 3 and detailed in Algorithm 1, starts with a single random-hypervector, uniformly sampled from the hyperspace of dimension d (generated by the function *random_hypervector(d)* in the algorithm). From there, inspired by the creation of the level-hypervectors, a sequence of transformations (T) are made to create $n/2$ level correlated hypervectors. Such transformations consist of XORing (also called *binding* in HDC, represented by the symbol $\oplus$) with what we name transformation-hypervectors ($\mathbf{t}$), which are placed in a queue (Q). The second half of hypervectors are then obtained by performing backward transformations (T^{-1}): the transformation-hypervectors are popped from Q (*first-in-first-out*) and sequentially bound to the current vector in order to generate the next one.

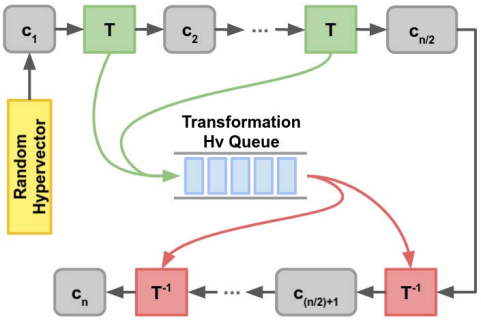

Figure 3. Illustration of the process to create circular-hypervectors. The curved arrows represent transformation-hypervectors being inserted in/removed from Q.

Algorithm 1: Creation of circular-hypervectors

Input : Two integers n and d.[1]
Output: A set $\{\mathbf{c}_1, \ldots, \mathbf{c}_n\}$ of n d-dimensional circular-hypervectors.

Define an empty queue Q `// Transformation Hv Queue`
$\mathbf{c}_1 \leftarrow$ random_hypervector(d)
`/* Perform forward transformations (T) */`
for $i \in \{2, \ldots, \frac{n}{2}\}$ **do**
  $\mathbf{t} \leftarrow \mathbf{0}^d$ `// d-dimensional zeros vector`
  Flip d/m random bits of $\mathbf{t}$
  $\mathbf{c}_i \leftarrow \mathbf{c}_{i-1} \oplus \mathbf{t}$
  Enqueue($Q, \mathbf{t}$)
`/* Perform backwards transformations (T^-1) */`
for $i \in \{\frac{n}{2} + 1, \ldots, n\}$ **do**
  $\mathbf{t} \leftarrow$ Dequeue(Q)
  $\mathbf{c}_i \leftarrow \mathbf{c}_{i-1} \oplus \mathbf{t}$
return $\{\mathbf{c}_1, \ldots, \mathbf{c}_n\}$

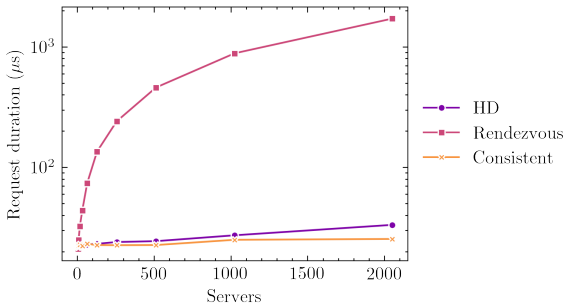

Figure 4. Average request handling duration as the number of servers in the pool increases.

5 Results and Discussion

5.1 Experimental setup

We have created a purpose build emulation framework to empirically verify our results. The emulator consists of two modules, a hash table and a generator. The generator emulates the requests from the outside world being sent to the hash table. The hash table module reads incoming requests from a buffer and uses a hashing algorithm to map them to an available server. Servers are added and removed using two special case requests, a `join` and `leave` request, respectively, with a unique identifier of the server. This functional emulator can be used to determine the computational efficiency of various hashing algorithms as well as their robustness to memory errors as we will describe next.

Since we do not have access to specialized HDC hardware and building the hardware is outside the scope of our work we had to implement the HDC operations using commodity hardware. To closely match the parallel nature of HDC hardware, we decided to implement HDC operations on a GPU. We used an Nvidia TITAN Xp GPU with 3840 cores and 12 GB of memory. The GPU's communication overhead was reduced by performing mappings in batches of 256 requests.

Each test was performed with different numbers of servers in the pool, going up to 2048. This scale is enough to show the results and trends of interest, but it is important to emphasize that like the other methods HD hashing can scale to much larger clusters, and even be used hierarchically (standard way to scale such hashing systems [20, 24]) to handle extremely high numbers of servers.

5.2 Efficiency

We executed each hashing function in our emulator to empirically determine its computational efficiency. First the generator sends n `join` requests to add available servers to the hash table module. Then, the generator sends 10,000 requests and tracks the wall-time. From this we determine the average time to handle a request.

[1]For ease of understanding, this version assumes that n is even. To generate a set of odd cardinality of circular-hypervectors, simply generate $2n$ and return just $\{\mathbf{c}_1, \mathbf{c}_3, \mathbf{c}_5, \ldots, \mathbf{c}_{2n}\}$.

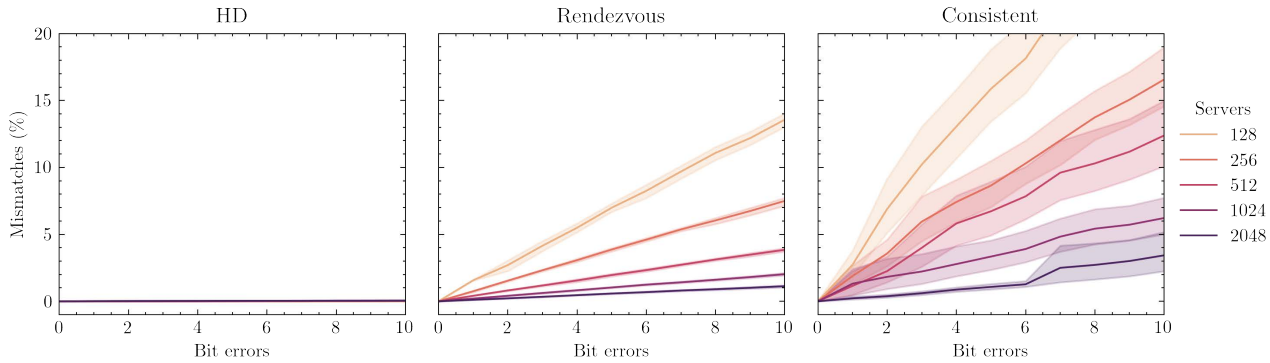

Figure 5. Percentage of mismatched requests when a number of bit errors occur.

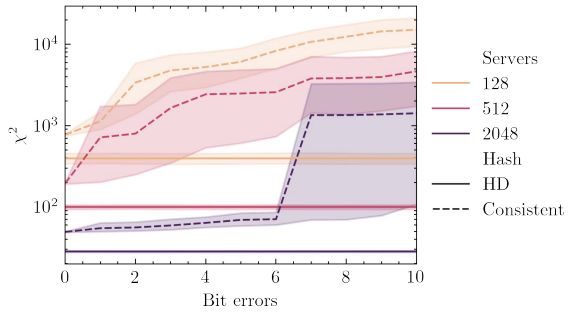

Figure 6. The discrepancy between the distribution of requests per server obtained by each algorithm and the uniform distribution, for different numbers of servers and bit errors, measured with the Pearson's χ^2 statistical test.

For various numbers of n, ranging from 2 to 2048 in powers of 2 the results are shown in Figure 4. The $O(n)$ time complexity of rendezvous hashing is clearly evident as is the superior computational efficiency of consistent hashing with respect to rendezvous hashing. Our HDC implementation using commodity hardware has a very similar scaling profile to consistent hashing. This confirms our belief that HDC hardware can appropriately be simulated by a GPU. However, as highlighted in Section 3, we expect the use of HDC accelerators to reduce the request handling time to a constant with the extreme of a single clock-cycle.

5.3 Robustness

As motivated before, the other main goal of HD hashing is to be a robust alternative to consistent and rendezvous hashing. In order to assess the performance of each hashing algorithm in an environment subject to noise, two experiments were performed using the emulator's noise injection capabilities. The first and most important, whose results are in Figure 5, shows how the ability of each technique to map keys to the correct value degrades when a certain number of bits in memory are randomly flipped. Ibe et al. [6] show that for 22 nm technology, 4-bit and 8-bit bursts occur 10% and 1% of the time, respectively. Moreover, errors within a machine are found to be strongly correlated, if a machine experienced an error it is 13-228 times more likely to experience another error in the same month [19]. To capture such features of a realistic scenario, we test each hashing technique in the range of 0 to 10 bit flips.

In our experiments, HD hashing confirmed our expectations, turning out to be far superior as none of the requests sent were matched to the wrong server. Meanwhile, in both consistent and rendezvous hashing an increasing percentage of mismatches occur, depending on the noise level.

In the second experiment we tested how uniform the distribution of requests among servers is and how uniform they remain when bits of the hash values are randomly inverted. For evaluation, we used the following *Pearson's chi-squared test* [4] to measure goodness of fit between our observed frequency distribution and the uniform distribution:

$$\chi^2 = \sum_{s_i \in S} \frac{\left(R\left(s_i\right) - E\right)^2}{E}$$

where $R(s_i)$ is the number of requests mapped to server s_i by the algorithm and $E = \frac{|R|}{|S|}$ is the uniformity expectation where $|R|$ and $|S|$ are the total number of requests and servers, respectively. The results, illustrated in Figure 6, show that not only does HD hashing distribute requests more uniformly than consistent hashing in an ideal scenario, but also that the presence of bit errors worsens the uniformity of consistent hashing even more, while that of HD hashing remains intact. To make the plot more readable, we omit the rendezvous hashing result. Note, from the description of the algorithm in Section 2.2, that rendezvous hashing is based only on the output of the hash function, that is, a pseudo-random number. Therefore, its assignment is perfectly (pseudo-) uniform and is not affected by bit errors. Rendezvous hashing, however, still suffers from mismatches and the method has less applicability due to its lower efficiency as illustrated in Figures 5 and 4, respectively.

6 Future work

Besides being a central component of our work, circular-hypervectors provide a way to represent periodic information that has not been available in the HDC literature thus far. Consider, for example, the seasons of the year, clearly there is a periodic relationship between them. Several other time-related examples can be listed such as hours of a day or days of a week, as well as other angular data such as directions, geolocation or color spaces. Whether this can be used to improve data representation in HDC, for instance in machine learning applications, is a promising direction of future work.

Our method can utilize the work by Schmuck et al. [18] that shows how HDC accelerators can optimize server lookup (inference in HDC) to a single clock-cycle. Realizing an implementation of the HD hashing algorithm in special hardware is future work.

7 Conclusion

We propose Hyperdimensional (HD) hashing—a novel algorithm based on Hyperdimensional Computing (HDC) which allows dynamic scaling of the hash table with minimal rehashing, a problem found in some of the most popular web applications. Through an emulation framework, we compare our method with consistent and rendezvous hashing and the experimental results show that HD hashing is the only approach that guarantees both efficiency and robustness. HD hashing scales similar to consistent hashing, while both are significantly more efficient than rendezvous hashing. Consistent hashing suffers from more than 20% mismatches with a realistic level of memory errors, which are common in large-scale cloud systems, while HD hashing remains unaffected. This superior level of tolerance to bit errors reduces the chance of critical failures in load balancing and web caching systems, among others.

Acknowledgment

This work was supported in part by a UCI Seed grant for Artificial Intelligence Research for Precision Health.

References

[1] Yahya Al-Dhuraibi et al. 2017. Elasticity in cloud computing: state of the art and research challenges. *IEEE Transactions on Services Computing* 11, 2 (2017), 430–447.

[2] Giuseppe DeCandia et al. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS Operating Systems Review* 41, 6 (2007), 205–220.

[3] Daniel E Eisenbud et al. 2016. Maglev: A fast and reliable software network load balancer. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 523–535.

[4] Priscilla E Greenwood and Michael S Nikulin. 1996. *A guide to chi-squared testing*. Vol. 280. John Wiley & Sons.

[5] Andy A Hwang et al. 2012. Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design. *ACM SIGPLAN Notices* 47, 4 (2012), 111–122.

[6] Eishi Ibe et al. 2010. Impact of scaling on neutron-induced soft error in SRAMs from a 250 nm to a 22 nm design rule. *IEEE Transactions on Electron Devices* 57, 7 (2010), 1527–1538.

[7] Mohsen Imani et al. 2017. Ultra-efficient processing in-memory for data intensive applications. In *Design Automation Conference (DAC)*. IEEE, 1–6.

[8] Mohsen Imani et al. 2017. Voicehd: Hyperdimensional computing for efficient speech recognition. In *International Conference on Rebooting Computing (ICRC)*. IEEE, 1–8.

[9] Pentti Kanerva. 2009. Hyperdimensional computing: An introduction to computing in distributed representation with high-dimensional random vectors. *Cognitive computation* 1, 2 (2009), 139–159.

[10] David Karger et al. 1997. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing (STOC)*. 654–663.

[11] Haitong Li et al. 2016. Hyperdimensional computing with 3D VRRAM in-memory kernels: Device-architecture co-design for energy-efficient, error-resilient language recognition. In *International Electron Devices Meeting (IEDM)*. IEEE, 16–1.

[12] Alec Xavier Manabat et al. 2019. Performance analysis of hyperdimensional computing for character recognition. In *International Symposium on Multimedia and Communication Technology (ISMAC)*. IEEE, 1–5.

[13] Vahab Mirrokni et al. 2018. Consistent hashing with bounded loads. In *Symposium on Discrete Algorithms (SIAM)*. SIAM, 587–604.

[14] Fateme Rasti Najafabadi et al. 2016. Hyperdimensional computing for text classification. In *Design, Automation Test in Europe Conference Exhibition (DATE), University Booth*. 1–1.

[15] Erik Nygren et al. 2010. The akamai network: a platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review* 44, 3 (2010), 2–19.

[16] Abbas Rahimi et al. 2016. Hyperdimensional biosignal processing: A case study for EMG-based hand gesture recognition. In *International Conference on Rebooting Computing (ICRC)*. IEEE, 1–8.

[17] Abbas Rahimi et al. 2017. High-dimensional computing as a nanoscalable paradigm. *Transactions on Circuits and Systems I: Regular Papers* 64, 9 (2017), 2508–2521.

[18] Manuel Schmuck et al. 2019. Hardware optimizations of dense binary hyperdimensional computing: Rematerialization of hypervectors, binarized bundling, and combinational associative memory. *Journal on Emerging Technologies in Computing Systems (JETC)* 15, 4 (2019), 1–25.

[19] Bianca Schroeder et al. 2009. DRAM errors in the wild: a large-scale field study. *ACM SIGMETRICS Performance Evaluation Review (PER)* 37, 1 (2009), 193–204.

[20] Haiying Shen et al. 2006. Cycloid: A constant-degree and lookup-efficient P2P overlay network. *Performance Evaluation* 63, 3 (2006), 195–216.

[21] Vilas Sridharan and Dean Liberty. 2012. A study of DRAM failures in the field. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 1–11.

[22] Ion Stoica et al. 2003. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking* 11, 1 (2003), 17–32.

[23] David G Thaler and Chinya V Ravishankar. 1998. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking* 6, 1 (1998), 1–14.

[24] Wei Wang and Chinya V Ravishankar. 2009. Hash-based virtual hierarchies for scalable location service in mobile ad-hoc networks. *Mobile Networks and Applications* 14, 5 (2009), 625–637.

[25] Tony F Wu et al. 2018. Brain-inspired computing exploiting carbon nanotube FETs and resistive RAM: Hyperdimensional computing case study. In *International Solid-State Circuits Conference (ISSCC)*. IEEE, 492–494.