LETTER

H-TLA: Hybrid-Based and Two-Level Addressing Architecture for IoT Devices and Services

Sangwon SEO[†a)], ***Member***, **Sangbae YUN**[††], **Jaehong KIM**[†], **Inkyo KIM**[†††], **Seongwook JIN**[††††], ***and*** **Seungryoul MAENG**[†], ***Nonmembers***

SUMMARY An increasing number of IoT devices are being introduced to the market in many industries, and the number of devices is expected to exceed billions in the near future. With this trend, many researchers have proposed new architectures to manage IoT devices, but the proposed architecture requires a huge memory footprint and computation overheads to look-up billions of devices. This paper proposes a hybrid hashing architecture called H- TLA to solve the problem from an architectural point of view, instead of modifying a hashing algorithm or designing a new one. We implemented a prototype system that shows about a 30% increase in performance while conserving uniformity. Therefore, we show an efficient architecture-level approach for addressing billions of devices.

key words: *H-TLA, jump hash, consistent hashing, hash*

1. Introduction

An Increasing number of IoT devices are being introduced to many industries, and the number of devices is expected to exceed billions in the near future [4]. With this trend, a lot of researches and studies are being done on managing IoT devices effectively and efficiently [1]–[3], [7], [10]. One of the most important issues that need to be solved is processing the look-up quickly in the IoT platforms because the number of devices on the platform can be in the billions. For traditional non-IoT platforms, we used to store device-information in relational databases and search as needed. However, if there are billions of devices, the relational database easily becomes a bottleneck [5], and it will take a long time to search for even one device. To solve this problem, new techniques that can search a device quickly such as hashing were discussed. Among those, *Consistent Hashing* [6] and *Jump Hash* [8] stood out as two of the most efficient ways.

While consistent hashing operates effectively on IoT platforms that frequently add or delete their nodes, consistent hashing extremely consumes memory to create and manage many virtual buckets. Jump hash has been proposed to solve the high memory usage of consistent hashing. Jump hash minimizes the memory usage while keeping consistency to addition and deletion of nodes at the same level of consistent hashing. It also provides a high level of uniform distribution that is equivalent to that of consistent hashing but needs computations instead of memory. To calculate the hash, it makes statistics-based jumps that require a heavy computation to produce a hashed value.

Although jump hash has an advantage from a memory usage perspective, it requires still many computational resources. The computational overhead is high because a lot of jumps occur in the early stage. This is one of the characteristics of jump hash. During the early stage, the gaps between jumps are very dense. However, the interval of jumps becomes longer, as more jumps occur, thereby decreasing the computational overhead. If we can decrease the number of jumps in the early stage, we would be able to decrease the computational overhead while keeping the low memory usage. This can be an ideal hashing algorithm that can effectively manage billions of devices.

In this paper, we propose `H-TLA` (Hybrid-based and Two-Level addressing Architecture) that minimizes the computational overhead from jump hash. We focus on effectively exploiting jump hash as a look-up algorithm, instead of modifying and enhancing jump hash itself. `H-TLA` introduces *Hybrid Hash* to significantly lower the computational overhead. In the early stage, it lowers the computational overhead by performing a simple bit-wise hash instead of a computation-heavy jump hash. It uses jump hash in the later stages so that consistency can still be supported even in the case of a node addition or deletion.

`H-TLA` manages all the nodes as 2-dimension of buckets and nodes, instead of 1-dimension. Since there are multiple nodes in a bucket, both bucket hash and node hash are required to compute a single hash value, which is called *Two-level Addressing*. With the two-level addressing, `H-TLA` can reduce the addressing time because two-level addressing cut down the early stage of jump hash significantly, which is a performance bottleneck of `H-TLA`

In this research, we conducted our experiments focusing on the performance and uniformity. We implemented the prototype of `H-TLA` based on jump hash. The experiment results show `H-TLA` has a low memory usage and computational overhead while providing consistency. Especially, `H-TLA` shows 40% better performance compared to jump hash, by effectively lowering the computational overhead of the jump hash. The results show a small decline in uniformity, which is only about 0.2%.

Manuscript received February 16, 2020.
Manuscript revised April 16, 2020.
Manuscript publicized May 14, 2020.
[†]The authors are with Computer Science Department, KAIST, Korea.
[††]The author is with Pinplay, Korea.
[†††]The author is with Samsung Electronics, Korea.
[††††]The author is with KIWI PLUS, Korea.
a) E-mail: smiler.seo@gmail.com
DOI: 10.1587/transinf.2020EDL8027

2. Background and Motivation

While *Consistent Hashing* [6] provides consistency by leveraging virtual buckets, it has the problem of high memory usage. *Jump Hash* [8] uses computations to provide consistency with low memory consumption. The jump refers to the computations to find the node where a key belongs. Consuming the computational resources instead of the memory resources is the unique characteristic of jump hash.

Based on the probability of a key being assigned to a specific node, Jump hash continuously makes jumps until the total distance of jumps becomes greater than the total number of nodes. The probability starts from 100% when the node size is 1 and decreases to $1/n$ when the total number of nodes is n. This is a harmonic series and its probability decreases dramatically when the number of nodes increases.

As described in Fig. 1, the jumps are more dense in the early stage and more sparse in the later stage. In jump hash, the jumps are connected to the computational overhead. There is no relationship between the jump distance and the overhead, but the overhead is related to the number of jumps. As the number of nodes increases, the number of jumps and the overall computations also increase. However, as the jump distance becomes larger, the frequency of the jump becomes relatively sparse. As a result, as the total number of nodes increases, the required computational overhead per node dramatically decreases.

Figure 2 describes the total elapsed time of jump hash for 1 billion keys as the number of nodes increases. It shows that the computational overhead is considerably high even when the number of nodes is small. Moreover, when the number of nodes increases by 10 times from 100 to 1,000, the time to execute increases only by 20%, not by 10 times. As the number of nodes increases, the computational overhead increases at a slower rate. In the early stage, the jumps are short and happen very frequently because the probability is high. However, in the later stage, the jumps are long and sparse due to the low probability, decreasing the computational overhead.

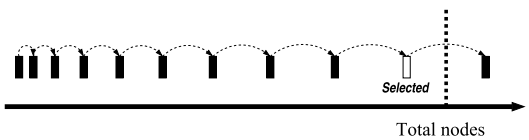

Fig. 1 Basic operation of jump hash

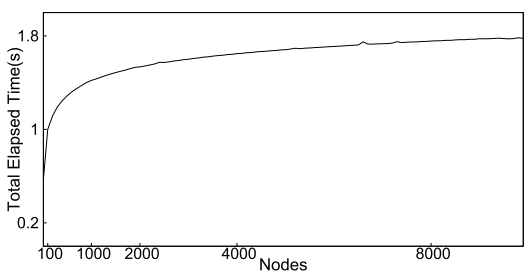

Fig. 2 Total elapsed time of jump hash

A lot of jumps occur in the early stage is one of the characteristics of jump hash. If we can decrease the number of jumps in the early stage, we would be able to decrease the computational overhead while keeping the low memory usage. This can be an ideal hashing algorithm that can effectively manage tens of billions of keys.

3. Design

3.1 Hybrid Hash

In this work, we attempt to solve the issue from an architectural point of view, instead of directly modifying jump hash or designing a new algorithm. To solve the computational overhead problem, we propose *Hybrid Hash* as an architectural approach that has two steps: bit-wise hash and jump hash. Bit-wise hash doesn't provide consistency feature but has a much lower computational overhead than jump hash because of the simple computations. Hybrid hash uses the existing jump hash without any modification. The difference from using only jump hash is as follows. In the early stage where there are a lot of jumps causing the computational overhead, jump hash is replaced by bit-wise hash. Jump hash which supports consistency is used for the rest. By using this mechanism, we can eliminate the computational overhead in the early stage while providing consistency feature to minimize the redistribution of keys when the number of nodes increases.

Figure 3 describes regular jump hash and hybrid hash. In jump hash, jumps are dense in the beginning and become more sparse as it gets closer to the total number of nodes, which explains why there is more computational overhead in the beginning. On the other hand, bit-wise hash is used between from node 1 to node 256. From node 257 jump hash is used. By starting jump hash from the node 257 instead of the node 1, the distance of each jump becomes much longer and the entire nodes can be covered with much fewer jumps. This eliminates the computational overhead in the early stage. Theoretically, for the first 256 nodes, 6.12 jumps are required on average. On the other hand, from node 257 to 1024, the average number of jumps is 1.38. While the number of nodes in the latter is four times bigger, the number of jumps is only 22.5% of the former. This clearly shows that replacing jump hash with bit-wise hash in the early stage is very effective.

Bit-wise hash to minimize the computational overhead is effective, compared to jump hash. Although both the performance and uniformity characteristics of bit-wise hash are very good, it is unable to provide consistency when a node is added or deleted, which is a critical disadvantage for the IoT platforms. For example, when the node size increases, bit-wise hash needs to redistribute the entire keys, which causes a serious computational overhead. In this paper, bit-wise hash is only used for the pre-determined, the early stage, which doesn't require consistency. The rest of the stage is covered by jump hash. One disadvantage of using bit-wise

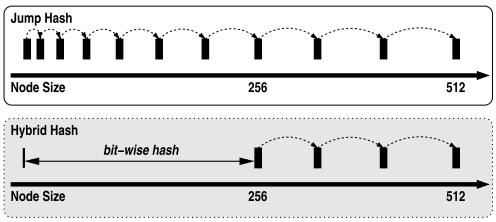

Fig. 3 Jump hash and hybrid hash

hash in the early stage is that it can't be changed once determined. However, the number of nodes for IoT platforms, which we focus on, is more than tens of thousands node in most cases and well beyond the number that bit-wise hash can cover. Instead of modification of jump hash, hybrid hash leveraging the architectural approach can be very effective.

3.2 Two-Level Addressing

`Two-level Addressing` consists of buckets and nodes. A key finds its bucket through a hash calculation. Then through another hash calculation, it finds the node it belongs to within the bucket. Through the first hash calculation, all requests are distributed evenly to 1/bucket and then again to the number of nodes in each bucket through the second hash calculation.

An advantage for two-level addressing is that it can reduce the addressing time. The bucket hash is used to find the right bucket and the node hash is needed to find the node within the bucket. One might assume that it would require more time since there are two levels of hash instead of one. However, if the bucket hash and the node hash can be done in parallel, we can reduce the overall time. The key point is that the node hash doesn't have to wait until the bucket hash is completed. To perform a precise node hash, we need to know the total number of nodes in a certain bucket through a bucket hash calculation. Regardless, `H-TLA` starts both hashes at the same time. A node hash calculation is performed until we know the total number of nodes in a bucket. When the bucket hash is completed, we can determine whether to continue the node hash based on the total number of nodes. However, as shown in Fig. 4, if the bucket hash takes long, the jumps can exceed the total number of nodes. In this case, `H-TLA` disregards the redundant computations and use the results reflecting the total number of nodes.

3.3 Uniformity

The uniformity is as important as the performance in hashing. It is difficult to use a hashing algorithm with a low uniformity, even if it provides high performance. `H-TLA` provides some enhancements from a uniformity perspective. In the case of hybrid hash, bit-wise hash is used until it reaches a fixed size, and after that, jump hash is used. Although the uniformity of bit-wise hash is lower than that of jump hash, the impact on bit-wise hash is reduced to a level similar to jump hash. Unlike hybrid hash, two-level addressing can impact the uniformity. Two-level addressing adds another

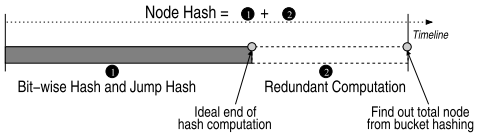

Fig. 4 Redundant computation on two-level addressing

hash layer on top of the current one-level jump hash, and therefore, the uniformity errors of jump hash will be duplicated.

The number of keys is denoted by k and the node size is denoted by n. When k becomes big enough, the uniformity of jump hash increases. On the other hand, when n increases, the uniformity decreases. The uniformity of the one-level jump hash is proportional to k/n. In the case of the two-level jump hash, we need to consider the buckets. The number of buckets is denoted by b. Two-level jump hash distributes keys (k) evenly to buckets (b), which can be expressed as k/b. Then k/b keys are distributed to n/b nodes in each bucket, which can be expressed as $k/b * ((k/b)/(n/b))$. This can also be expressed as $k/b * k/n$. This represents the uniformity of one-level jump hash. The uniformity of two-level hash changes depending on how k/b changes. In other words, the uniformity of two-level addressing is proportional to k and inversely proportional to b. The uniformity of two-level addressing would be lower than that of one-level addressing since there is an additional jump hash. But the degree of such decrease depends on the number of buckets, which is discussed in Sect. 4.2

4. Results

We evaluated `H-TLA` on AMD Ryzen 7 2700 processors with physical 8 cores. The processor has 768KB L1 cache, 4MB L2 cache, and 16MB L3 cache. The machine has 16GB memory and uses Ubuntu Linux 18.04 as the operating system. To evaluate the effectiveness of `H-TLA`, we created a simple workload which generates random keys and runs hash operations for the keys. The workload is designed to measure the throughput and uniformity of `H-TLA`. For bit-wise hash, we adopt Pearson hashing [9] designed for fast execution.

4.1 Performance Evaluation

Figure 5 describes the performance difference between jump hash and two-level addressing. In the figure, *jump hash* represents jump hash and *Hybrid Hash* represents hybrid hash. *Two-level* represents two-level addressing using hybrid hash. We used a 128 bit-wise hash and distributed 10 billion keys to 1 million nodes.

Figure 5 also shows the performance improvement through two-level addressing. Two-level addressing performs better than a single hybrid hash in the figure because the early stages of jump hash were significantly reduced by using two-level addressing. For example, a single hybrid hash can apply fast bit-wise hash once, whereas two-level

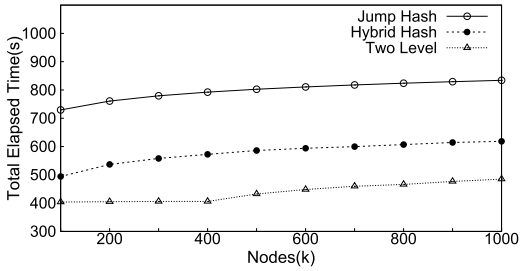

Fig. 5 Performance of H-TLA

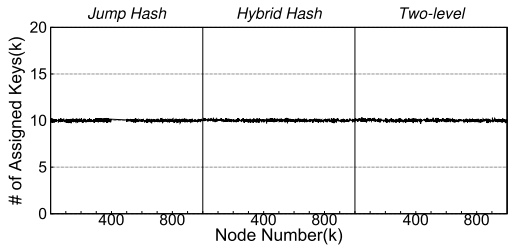

Fig. 6 Uniformity comparison

addressing can apply double times. If the total number of nodes goes beyond a certain threshold, the performance enhancement is limited because the early stages can't be reduced anymore. In contrast, H-TLA can not provide a consistency feature if the number of nodes is smaller than that of the early stage. H-TLA shows no difference in performance compared to the jump hash when the size of early-stage is reduced to provide consistency.

4.2 Uniformity Evaluation

For hashing, the uniformity is as important as the performance. Figure 6 consists of three charts that show the uniformity of each configuration. In this experiment, one million nodes and 10 billion keys were used. Each key has a random and unique number used for hashing. Ten thousand keys must be allocated to each node for the perfect uniformity.

Jump Hash: The left part of the chart shows the uniformity of the jump hash. The top 90% of the nodes received the most keys which are 10,212 keys. On the other hand, the bottom 90% received 9,870 keys. We can say that the jump hash has an error of about 2%.

Hybrid Hash: The middle chart shows the uniformity of the hybrid hash. A hybrid hash uses 128 bits for the Bitwise hash. For the top 90%, the hybrid hash received 10,142 keys. For the bottom 90%, the hybrid hash received 9,864 keys. As shown in the chart, there is not much difference between the jump hash and the hybrid hash.

Two-level Addressing: The right part of the chart shows the uniformity of two-level addressing. The two-level addressing uses 128 bits and performs the hybrid hash twice. The top 90% with most keys from the two-level addressing received 10,285 keys. The bottom 90% received 9,786 keys. We can see that the uniformity of the two-level addressing is slightly worse than that of a single hybrid hash because the error accumulates through an additional hash after the first hybrid hash. However, adding another level to a single hybrid hash to a two-level addressing only increases the errors by 0.5%, not by 2 times.

5. Conclusions and Future Work

As more and more IoT devices will impact our lives, it is crucial to manage IoT devices efficiently. To address this, we proposed a new addressing architecture that solves the computational overhead problem of jump hash. Specifically, it enhances the computational overhead issues through an architecture, not through a performance enhancement by modifying the jump hash algorithm. The experiment showed a 30% better performance than that of the jump hash algorithm with a similar level of uniformity. we show an efficient architecture-level approach for managing billions of devices.

Beyond effective addressing of billions of devices, IoT platforms need management-related features. If a certain node fails after billions of files are distributed to many nodes, it should be quickly replaced with alternative nodes. We will extend `H-TLA` to provide a weighted load balancing and fail-over features, leveraging the characteristics of the jump hash algorithm. Considering additions, deletions, and connections of devices on the fly, IoT platforms need to preserve the status of billions of devices, which requires a huge computation and memory. If each node has its own status map for their devices, we can simply find the status of a specific device through two-level addressing of `H-TLA`. We will research an efficient architecture of a distributed map with `H-TLA`.

References

[1] M. Aazam and E.-N. Huh, "Fog computing and smart gateway based communication for cloud of things," 2014 International Conference on Future Internet of Things and Cloud, pp.464–470, Aug. 2014.
[2] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of things: A survey on enabling technologies, protocols, and applications," IEEE Commun. Surv. Tutorials, vol.17, no.4, pp.2347–2376, 2015.
[3] J. Gubbi, R. Buyya, S. Marusic, and M. Palaniswami, "Internet of things (IoT): A vision, architectural elements, and future directions," Future Generation Computer Systems, vol.29, no.7, pp.1645–1660, 2013.
[4] Internet of things, https://en.wikipedia.org/wiki/Internet_of_things.
[5] J. Han, H.E,G. Le, and J. Du, "Survey on NoSQL database," 2011 6th International Conference on Pervasive Computing and Applications, pp.363–366, Oct. 2011.
[6] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," Proc. Twenty-ninth annual ACM symposium on Theory of computing - STOC '97, pp.654–663, New York, NY, USA, 1997.
[7] R. Khan, S.U. Khan, R. Zaheer, and S. Khan, "Future internet: The internet of things architecture, possible applications and key challenges," 2012 10th International Conference on Frontiers of Information Technology, pp.257–260, 2012.
[8] J. Lamping and E. Veach, "A fast, minimal memory, consistent hash algorithm," CoRR, abs/1406.2294, 2014.

[9] P.K. Pearson, "Fast hashing of variable-length text strings," Commun. ACM, vol.33, no.6, pp.677–680, June 1990.

[10] P. Sethi and S.R. Sarangi, "Internet of things: Architectures, protocols, and applications," Journal of Electrical and Computer Engineering, vol.2017, pp.1–25, 2017.