Fast Robust Hashing

Manuel Urueña, David Larrabeiti and Pablo Serrano
Universidad Carlos III de Madrid
E-28911 Leganés (Madrid), Spain
Email: {muruenya,dlarra,pablo}@it.uc3m.es

***Abstract*— As stateful flow-aware services are becoming commonplace, distributed router architectures have to quickly assign packets being forwarded to service-specialized processors in order to balance flow processing and state among them. Moreover, packets belonging to the same flow must be always assigned to the same CPU, even if some of the service processors become unavailable. This paper presents two novel Fast Robust Hashing algorithms for persistent Flow-to-CPU mapping, that require less hashing operations per packet than previous Robust Hash algorithms, thus being able to fulfill all the above requirements to implement flow-aware services at wire-speed.**

***Index Terms*— Robust Hashing, Flow Processing**

I. Introduction

Providing flow-aware services for hundreds of thousands of flows per second arriving through high-speed interfaces poses significant technical challenges. A common solution is to employ distributed systems that allows parallel packet processing. These multi-processor router architectures [8] allow flow state to be distributed among specialized CPUs, or even among external devices. For example deploying a cluster of NAT appliances attached to the access router of an organization is quite common nowadays.

In any case, those stateful services require some kind of mapping mechanism that does not only balance flows among CPUs or external devices, but it must also ensure that all packets belonging to the same flow are always processed by the same CPU. Otherwise, for example in the case of a stateful firewall, any mismatched packets will be dropped as only the initial CPU is aware of such connection.

A mapping scheme based on hash functions seems ideal for distributed packet processing, as it does not require a central flow table, that could easily become the bottleneck of the system. Instead it is only necessary to perform one hash operation to the packet flow identifier in order to find the CPU that must process all the packets belonging to that flow.

However, this scheme fails when the number of available processors varies, for example when one of the NAT boxes in the cluster goes down. In that case, the return range of the hash function should be reduced by one in order to skip the disabled processor. But changing the hash function means that the Flow-to-CPU mapping obtained by the old hash function will differ from the one being performed by the new hash function, leading to the disruption of all the re-mapped flows. In fact, whenever any of the n CPUs goes down, most of the flows ($\frac{n-1}{n}$) [7] will be re-mapped (and therefore discarded), due to the load-balancing property of hashing.

The Robust Hashing mechanism was designed to address the above issue. Its objective is that, whenever a processor goes down, the only re-mapped flows are the ones that would be assigned to the disabled CPU ($\frac{1}{n}$). The solution is quite simple: instead of performing a single hash operation that returns the chosen CPU, each CPU has an associated hash function that gives its "affinity" to that flow. Then, the CPU with the greatest affinity value is chosen. When a CPU goes down its hashing function is not computed, thus it cannot be chosen. Therefore, only the flows assigned to it are re-mapped because, by definition, the rest of flows have a greater affinity with one of the remaining CPUs.

The term Robust Hashing was firstly introduced by Ross in [6] for the Web-caching domain. Later, the Highest Random Weight algorithm [7] by Thaler and Ravishankar enhanced the Robust Hash mechanism to support heterogeneous caches, by assigning them static weights. Finally, Kencl and Le Boudec [3] have defined a method for dynamically adjusting the weights associated to each CPU depending on its workload.

A drawback of all these techniques is that each mapping requires as many hash operations as CPUs are available. Because hash functions are quite complex operations and all these computations should be performed for each arriving packet, this load-balancing process could become the bottleneck of the whole system, thus limiting the scalability of the service-processor cluster.

This paper presents two novel algorithms, designed to reduce the number of hash computations per packet while keeping the same useful properties of the *classical* Robust Hash techniques: the Small Robust Hash and the Big Robust Hash algorithms.

II. Small Robust Hashing

The objective of the Small Robust Hash algorithm is to assign a packet to a CPU with just one hash operation. Therefore it employs a single hash function that never changes, even when a CPU is disabled, in order to maintain previous mappings.

The algorithm works as follows: when a packet arrives, its flow identifier f is employed as the argument of the hash function. However, this hash operation $h(f)$ does not return the CPU itself but a position in the "Mapping Vector", which contains the selected CPU.

In order to achieve probabilistic load balancing [7], the Mapping Vector must contain the same number of entries for each available CPU. Therefore, if a CPU goes down, all its

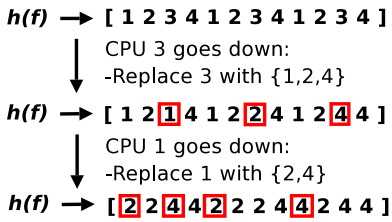

Fig. 1. Evolution of the Mapping Vector for the Small Robust Hash algorithm ($n = 4$, CPUs 3 and 1 are disabled)

entries in the Mapping Vector are renamed with the remaining CPUs in a round-robin fashion.

For example, for the Mapping Vector in Fig. 1 and employing $h(f) = f \mod 12$ as a simple hash function, all packets of flow 17 will be assigned to CPU 2, even if CPUs 1 and 3 go down. On the other hand, flows with identifier 14 start being processed by CPU 3, but later ones will be assigned to CPU 1 and then to CPU 4 when CPUs 3 and 1 go down, respectively.

A. Memory requirements

Small Robust Hash algorithm requires a Mapping Vector of length m such that, for any number of disabled CPUs, there should be an equal share of entries for each of the available CPUs left.

This requirement is quite strong, as it means that, if n is the total number of CPUs, m should be a multiple of n, $(n-1)$, ..., that is, all the possible number of available CPUs. Otherwise if m is not a multiple of the number of available CPUs, it is not possible to rewrite all the disabled CPU's entries with a fair round-robin, leading to some degree of load imbalance among the remaining CPUs.

Therefore, $m = \prod p^i$ where $p^i \leq n$ is the greatest i power of every prime number $p \in [2, n]$.

As it could be seen on the first row of Table I, m could be a huge value but for a relatively small number of CPUs. However this is only necessary in order to support an arbitrary number of CPUs going down, which is quite an improbable situation. Therefore it is possible to save some memory if only a subset of values between 2 and n is chosen, thus providing partial fault-tolerance just for a reasonable number of disabled CPUs. The second and third rows of Table I show the memory requirements of the Small Robust Hash algorithm in order to support a CPU failure rate of 25% and 10% without load-imbalance among the surviving ones.

B. Performance

The Small Robust Hash algorithm poses an optimal performance, as it always finds each packet-to-CPU mapping with just one hash operation plus a memory access, no matter how many CPUs are down. Thus its complexity is $O(1)$.

Also, when a CPU is disabled, all its $\frac{m}{n-k}$ entries at the Mapping Vector should be found and replaced by available ones, being $n-k$ the number of CPUs left. Thus, the complexity to update the Mapping Vector is $O(m)$.

TABLE I
MEMORY REQUIREMENTS (IN BYTES)

Number of CPUs (n)	**4**	**8**	**16**	**32**
Small Robust Hash 100%	12	840	$7 \cdot 10^5$	$14 \cdot 10^{13}$
Small Robust Hash 25%	12	168	21840	$17 \cdot 10^8$
Small Robust Hash 10%	12	56	1680	$4 \cdot 10^5$
Big Robust Hash	10	36	136	528

However, the memory requirements of the Small Robust Hash algorithm make it applicable just for systems allowing partial fault-tolerance or having a small number of CPUs, hence its name. For distributed architectures with a greater number of processors, an algorithm with less memory requirements is proposed in the next section.

III. BIG ROBUST HASHING

The Big Robust Hash algorithm also applies hash functions to the packet's flow identifier in order to obtain a Mapping Vector's index. However, the Big Robust Hashing algorithm does not employ a single hash function but several ones, each one associated to a different Mapping Vector. All the Mapping Vectors together are called the "Mapping Matrix".

When all CPUs are active, this algorithm resembles the Small Robust Hashing one: there is one Mapping Vector (with a single entry per CPU) and a hash function that returns a position inside this vector. The differences only arise if some CPUs are disabled: the Big Robust Hash algorithm does not employ a single, long vector but many.

When a CPU goes down, a new Mapping Vector containing the remaining CPU entries, and its associated hash function are added. Then, the disabled CPU entry at the initial vector is replaced by a pointer (called *hop*) to the second vector, as shown in Fig. 2.

Then, the Mapping process works as follows: given f, the flow identifier of the packet, the initial hash function $h(f)$ returns a position at the initial Mapping Vector. If that entry is an available CPU, the mapping is done. Else, when a *hop* is found, the following hash function should be applied to select a position in the new vector. In general, this process is repeated until an available CPU is found.

Although this mechanism employs several hash functions, no disruption occurs as they are always applied in the same order. Flows previously assigned to an available CPU will be mapped again by the same hash function because its entry at the top Mapping Vector remains unchanged. On the other hand, new flows that would be assigned to the disabled CPU by the first hash function will find the *hop* entry, and then the second hash function $h'(f)$ will assign them to one of the available CPUs at the new vector. Thus the remaining CPUs will absorb an equal share of the flows that would be otherwise assigned to the disabled CPU.

If a second CPU goes down, another hash function $h''(f)$ and its Mapping Vector are added, and the entries of the disabled CPU in all other vectors are renamed with *hop* entries pointing directly to the third Mapping Vector.

Notice that the new vector does not replace the previous one, but it is added at the bottom of the Mapping Matrix. The

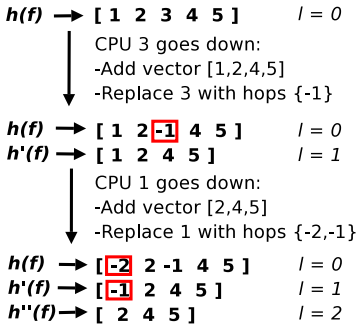

Fig. 2. Evolution of the Mapping Matrix for the Big Robust Hash algorithm ($n = 5$, CPUs 3 and 1 are disabled)

intermediate vectors and their hash functions are needed to avoid the disruption of the flows that were mapped employing them, right before the last CPU was disabled.

Let us illustrate the behavior of the Big Robust Hash algorithm with the aid of the Mapping Matrices of Fig. 2 and a simple family of hash functions based on the module operator ($h(f) = f \mod i$): $h(f) = f \mod 5$, $h'(f) = f \mod 4$, and so on. For example, flow 13 is always processed by CPU 4 because whenever the first hash function $h(13)$ is computed, it founds CPU 4 available every time. On the other hand, once CPUs 3 and 1 are disabled, packets with flow identifier 12 are mapped employing the three hash functions. $h(12)$ finds a -1 *hop* to the second Mapping vector and applying $h'(12)$ over the second vector has the same result. Finally $h''(12)$ assigns these packets to CPU 2. However packets of flow 10 will be classified just with two hash operations, because the -2 *hop* found at the first Mapping Vector redirects the search to the bottom one, which always succeeds ($h''(10) \mapsto$ CPU 4).

A. Memory requirements

As the Mapping Matrix is initialized with a Mapping Vector of length n, and a new one is added whenever a CPU goes down, the maximum number of Mapping Vectors is $n + 1$. Then, as the full Mapping Matrix is triangular, it only requires memory for storing $\frac{n(n+1)}{2}$ positions.

Table I shows, for different values of n, the maximum size of the Mapping Matrix required by the Big Robust Hash algorithm. Those values are small enough to fit in the data cache of commercial Network Processor [1], [2] micro-engines, thus memory access latency does not seem to be a major problem.

B. Performance

If $k \in [0, n]$ is the number of disabled CPUs, *classical* Robust Hash algorithms require $n - k$ hash operations in order to perform each flow-to-CPU mapping, that is, one hash operation per available CPU. That means that, when all CPUs are up an running, it is necessary to compute n hash operations per packet and choose the biggest value. On the other hand, both Fast Robust Hash algorithms presented in this paper require just one hash operation and a fast SRAM memory access per packet in the most common case, when all CPUs are available.

The complexity to update the Mapping Matrix when a CPU goes down is $O(k)$ because only one entry per Mapping Vector should be rewritten.

The Big Robust Hash algorithm performance study is less trivial than previous ones as, unlike the *classical* and Small variants, each packet is not always mapped after a fixed number of hash operations, but each one may require a different number of operations. Although, obviously, the maximum number of operations per packet is $k+1$, as this is the number of Mapping Vectors when k CPUs are down.

In order to study the mapping process, each vector of the Mapping Matrix is called *level*, and they are numbered in reverse order, from the bottom Mapping Vector to the top one, as shown in Fig. 2. This way, the level number $l \in [0, k]$ also indicates the number of disabled CPUs/*hop* entries at that level. Other useful functions that characterize a l level are:

$P_{CPUs}(l)$: Probability to find any available CPU at level l

$$P_{CPUs}(l) = \frac{n-k}{n-k+l} \tag{1}$$

$P_{hops}(l)$: Probability to find any *hop* entry at level l

$$P_{hops}(l) = 1 - P_{CPUs}(l) = \frac{l}{n-k+l} \tag{2}$$

All *hop* entries in a level are equiprobable (by the load-balancing property of the hash functions), thus $P_{hop}(l)$ is the probability to find one particular *hop* redirection at level l:

$$P_{hop}(l) = \frac{P_{hops}(l)}{l} = \frac{1}{n-k+l} \tag{3}$$

Let $H(l)$ be the random variable that defines the number of hash operations required to find an available CPU, starting with the hash function/Mapping Vector at level l. Then, we are interested in $\bar{H}(k)$, that is, the average number of hash operations until an available CPU is found, starting at the top of the mapping matrix ($l = k$).

The bottom level of the mapping matrix ($l = 0$) does not have any *hop* entries, thus only one hash operation is needed:

$$H(0) = 1 \Longrightarrow \bar{H}(0) = 1 \tag{4}$$

At upper levels ($l > 0$), the average number of hash operations is: 1 if a CPU entry is found ($P_{CPUs}(l)$), or 1 plus the average number of operations from a lower level ($\bar{H}(l-j)$) when the *hop* to that level is found. In every level there are l equiprobable *hop* entries, *each one pointing to a different lower level* (see level $l = 2$ at the bottom of Fig. 2), therefore:

$$\begin{aligned}
\bar{H}(1) &= P_{CPUs}(1) \cdot 1 + P_{hops}(1)(1 + \bar{H}(0)) \\
\bar{H}(2) &= P_{CPUs}(2) \cdot 1 + \frac{P_{hops}(2)}{2}(1 + \bar{H}(1)) + \\
&\quad \frac{P_{hops}(2)}{2}(1 + \bar{H}(0)) \\
\bar{H}(l) &= P_{CPUs}(l) + \frac{P_{hops}(l)}{l}(l + \sum_{i=0}^{l-1} \bar{H}(i)) \\
&= 1 + P_{hop}(l) \sum_{i=0}^{l-1} \bar{H}(i)
\end{aligned} \tag{5}$$

Particularizing it for the $l-1$ level:

$$\bar{H}(l-1) = 1 + P_{hop}(l-1)(\bar{H}(0) + ... + \bar{H}(l-2))$$
$$\bar{H}(0) + ... + \bar{H}(l-2) = \frac{\bar{H}(l-1)-1}{P_{hop}(l-1)} \quad (6)$$

Extending the elements of the summatory in equation (5) and replacing most of them with equation (6):

$$\begin{aligned}\bar{H}(l) &= 1 + P_{hop}(l)(\bar{H}(0) + ... + \bar{H}(l-2) + \bar{H}(l-1)) \\ &= 1 + P_{hop}(l)(\frac{\bar{H}(l-1)-1}{P_{hop}(l-1)} + \bar{H}(l-1)) \\ &= 1 + \frac{P_{hop}(l)}{P_{hop}(l-1)}(\bar{H}(l-1)-1) + P_{hop}(l)\bar{H}(l-1)\end{aligned} \quad (7)$$

Taking into account that:

$$\frac{P_{hop}(l)}{P_{hop}(l-1)} = \frac{\frac{1}{n-k+l}}{\frac{1}{n-k+l-1}} = 1 - \frac{1}{n-k+l} = 1 - P_{hop}(l) \quad (8)$$

From equations (7) and (8) we get:

$$\begin{aligned}\bar{H}(l) &= 1 + (1 - P_{hop}(l))(\bar{H}(l-1)-1) + P_{hop}(l)H(l-1) \\ &= \bar{H}(l-1) + P_{hop}(l)\end{aligned}$$

Extending the recursive elements of the formula:

$$\begin{aligned}\bar{H}(l) &= \bar{H}(l-1) + P_{hop}(l) \\ &= (\bar{H}(l-2) + P_{hop}(l-1)) + P_{hop}(l) \\ &= (\bar{H}(0) + P_{hop}(1)) + P_{hop}(2) + \cdots + P_{hop}(l)\end{aligned}$$

By (4) this arithmetic series could be summarized as:

$$\bar{H}(k) = 1 + \sum_{l=1}^{k} P_{hop}(l) = 1 + \sum_{l=1}^{k} \frac{1}{n-k+l} \quad (9)$$

Therefore, as we will see in Section IV, the average number of hash operations for the Big Robust Hash algorithm ($\bar{H}(k)$) is logarithmic with the number of disabled CPUs.

In order to characterize better the distribution of the number of hash operations per packet, we are also interested in the probability function $P_{hash}(h,l)$: Probability to perform h hash operations to find an available CPU, starting at level l.

The only way to perform a single hash operation is to find any of the available CPUs in the current level, thus:

$$P_{hash}(1,l) = P_{CPUs}(l)$$

In order to employ 2 hashes, the first hash query must return one of the l hops at that level, and the next query at a lower level $i \in [0, l-1]$ must succeed. Thus, the total probability is the sum of all the possible paths:

$$P_{hash}(2,l) = \sum_{i=0}^{l-1} P_{hop}(l) \cdot P_{hash}(1,i)$$

The reasoning for 3 hashes is the same, although in this case the *hop* to the bottom vector is forbidden, as it has no further *hops* and it is not possible to perform the 2 hash operations left. Thus, $i \in [1, l-1]$:

$$P_{hash}(3,l) = \sum_{i=1}^{l-1} P_{hop}(l) \cdot P_{hash}(2,i)$$

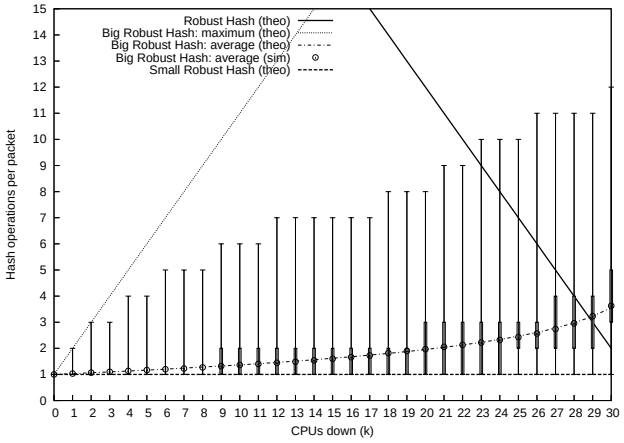

Fig. 3. Comparison of different Robust Hash algorithms ($n = 32$)

Therefore, as all hops are equiprobable, the general expression is:

$$P_{hash}(h,l) = \begin{cases} P_{CPUs}(l) & \text{if } h = 1 \\ P_{hop}(l) \sum_{i=h-2}^{l-1} P_{hash}(h-1,i) & \text{if } h \geq 2 \end{cases} \quad (10)$$

To validate these analytical results, Section IV compares the estimation of the average number of hash operations and the probability mass function of Big Robust Hash operations with trace-driven simulation measurements.

IV. Performance Evaluation

All simulations are based on a frame trace from a bi-directional Gigabit Ethernet link at the University of Purdue [5]. The trace spans 90 seconds and contains 18.7 million IP packets from 871 thousand distinct flows, with 933 Mbps and 210 thousand packets per second on average.

The flow identifier employed is the 32 bit integer built as the XOR of the source and destination IP addresses, protocol and TCP/UDP port numbers, with the protocol number and the greater port in the upper 16 bits and the other one in the lower 16 bits. Then a Fibonacci hash function [4], with the appropriate modulo, is employed to query the Mapping Vector.

Fig. 3 compares the performance of all the Robust Hash algorithms analyzed in this paper: *Classic* Robust Hash must execute $n-k$ hash operations per packet. Of course, the best performer is the Small Robust Hash algorithm which requires just one hash operation per packet irrespective of how many CPUs are down. However this result is purely theoretical, as the memory requirements for 32 CPUs with full fault-tolerance are far beyond from any practical implementation.

The Big Robust Hash is in a middle ground between them, as it trades-off performance for memory (528 Bytes for a 32 CPU cluster). Fig. 3 shows the theoretical curves of the average number of hash operations (Eq. (9)) and the $k+1$ upper limit of the Big Robust Hash algorithm. Regarding to the simulation measurements of the Big Robust Hash algorithm, the minimum, first quartile, mean, third quartile and the maximum number of hash operations employed per mapping are shown.

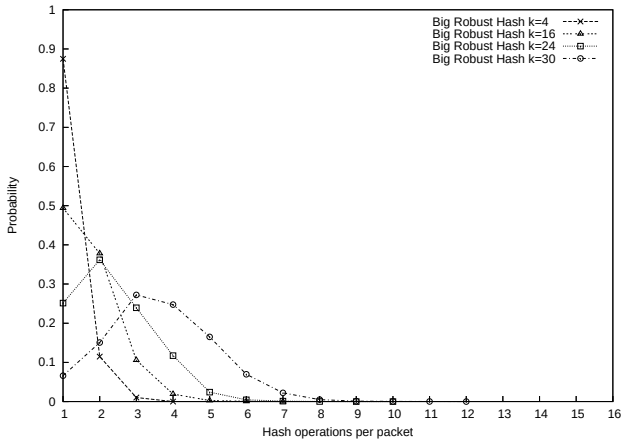

Fig. 4. Probability mass functions of Big Robust Hashing algorithm ($n = 32$)

As expected, the average number of operations increases much slower than the maximum values (i.e. with 31 Mapping Vectors, one of the 2 CPUs left is found only after 3.63 hash operations in average). This behavior could be better seen in Fig. 4 which shows, for representative values of k, the histograms with the number of hash operations employed by the Big Robust Hash simulation.

Table II shows the cumulative probability to perform h or less hash operations per packet for different values of n and k. These values have been obtained by evaluating the theoretical Eq. (10). For example, the fifth row ($n = 32, k = 16$) indicates that when only half of the 32 CPUs are available, the 99% of packets will be mapped with 4 or less hash operations, and most of them will require just 2 hashes (85%).

Another scalability property illustrated by Table II is that the number of hash operations does not depend on the total number of CPUs but on the available/total CPU ratio. For example, the cumulative probabilities when 50% CPUs are down (i.e. 2nd, 5th and 8th rows) are almost identical, in spite of the fact that the cluster size doubles in every step. Therefore, although increasing n implies a slight performance decrease, clearly the main factor is the $\frac{n-k}{n}$ ratio.

These results confirm that Big Robust Hash mappings does employ a very small number of hash operations when compared to *classical* Robust Hash algorithms. Moreover, the probability of performing more hash operations than the average value quickly tends to zero, as show in Fig. 4.

V. Conclusion

Nowadays, flow processing at edge routers is becoming commonplace (e.g. stateful firewalls, NAT, Session Border Controllers, etc.). Therefore, multi-processor routers need a fast, distributed, and scalable mechanism to load-balance flows among all their available service-specialized CPUs or to a cluster of external devices, while ensuring that all packets of a given flow are assigned to the same CPU. Robust Hash techniques are specially well suited to address this issue, although *classical* Robust Hash algorithms require several hash operations per packet in order to find which CPU is processing that flow.

TABLE II
Cumulative probability of the number of hash operations (h) for the Big Robust Hash algorithm

h	**1**	**2**	**3**	**4**	**5**
$n = 16, k = 4$	0.75	0.9738	0.9987	0.9999	1.0
$n = 16, k = 8$	0.5	0.8627	0.9771	0.9975	0.9998
$n = 16, k = 12$	0.25	0.6212	0.8694	0.9683	0.9944
$n = 32, k = 8$	0.75	0.9697	0.9978	0.9999	0.9999
$n = 32, k = 16$	0.5	0.8545	0.9720	0.9962	0.9996
$n = 32, k = 24$	0.25	0.6086	0.8531	0.9586	0.9909
$n = 64, k = 16$	0.75	0.9677	0.9973	0.9998	0.9999
$n = 64, k = 32$	0.5	0.8505	0.9694	0.9953	0.9994
$n = 64, k = 48$	0.25	0.6025	0.8449	0.9533	0.9887

This paper presents two Fast Robust Hash algorithms that require only one hash operation to perform each mapping when all CPUs are available, while *classical* Robust Hash algorithms do require one hash operation for each available CPU. These new algorithms achieve a better performance by employing Mapping Vectors to maintain a persistent mapping, even when several CPUs are disabled. The Small Robust Hash algorithm always finds the mapping with a single hash operation. However, when full fault-tolerance is required, it has severe memory requirements, thus it is only applicable to architectures with a small number of CPUs.

On the other hand, the Big Robust Hash algorithm does not employ a single, large Mapping Vector, but a small Mapping Matrix. In the worst case it requires one hash operation and a fast SRAM memory access for each disabled CPU, plus the initial ones. However, the average number of hash operations is far below this limit, as it is demonstrated with analytical and trace-driven simulation results. Therefore, the Fast Robust Hashing algorithms could be good design choices for cluster load-balancers or next generation multi-processor edge router architectures.

Acknowledgments

This work is being funded by the Spanish MEC under project IMPROVISA TSI2005-07384-C03. The authors wish also to thank Raquel Panadero, Iván Vidal, Ricardo Romeral, Carlos Jesús Bernardos for their valuable comments.

References

[1] J. Allen et al. *IBM PowerNP Network Processor: Hardware Software and Applications*. IBM Journal of Research and Development. Vol. 47, pp. 177-194, March/May 2003.
[2] Intel IXP2XXX Product Line of Network Processors: <http:// www.intel.com/design/network/products/npfamily/ixp2xxx.htm>
[3] L. Kencl and J. Y. Le Boudec. *Adaptive Load Sharing for Network Processors*. Proceedings of the IEEE INFOCOM 2002. Vol. 2, pp. 545-554, June 2002.
[4] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms* Second Edition. Addison-Wesley 1997.
[5] NLARN Trace from a Gigabit Ethernet link at the University of Purdue: <ftp://pma.nlanr.net/traces/daily/20060228/PUR-1141097090.erf.gz>. 28 February 2006.
[6] K. W. Ross. *Hash routing for collections of shared web caches*. IEEE Network, Vol. 11, No. 6, pp. 37-44, November/December 1997.
[7] D. G. Thaler and C. V. Ravishankar. *Using name-based mappings to increase hit rates*. IEEE/ACM Transactions on Networking, Vol. 6, No. 1, pp. 1-14, February 1998.
[8] L. Yang, R. Dantu, T. Anderson and R. Gopal. *Forwarding and Control Element Separation (ForCES) Framework*. RFC 3746, April 2004.