Flow Management at Multi-Gbps: Tradeoffs and Lessons Learned

Georges Nassopulos[1], Dario Rossi[1], Francesco Gringoli[2], Lorenzo Nava[2], Maurizio Dusi[3], and Pedro Maria Santiago del Rio[4,1]

[1] Telecom ParisTech, Paris, France
[2] Universita' degli Studi di Brescia, Brescia, Italy
[3] NEC Laboratories Europe, Heidelberg, Germany
[4] Universidad Autonoma de Madrid (UAM), Madrid, Spain

Abstract. While the ultimate goal of kernel-level network stacks is to manage individual packets at line rate, the goal of user-level network monitoring applications is instead to match packets with the flow they belong to, and take actions accordingly. With current improvements in Network Interface Cards hardware and network software stacks, traffic monitors and traffic analyzers are fed with multi-Gbps streams of packets – which de facto pushes bottlenecks from kernel-level networking stack up to user-level applications. In this paper, we argue that flow management is a crucial module for any user-application that needs to process traffic at multiple Gbps, and we study the performance impact of different design choices of the flow management module by adopting a trace-driven emulation approach. While our results do not show a single "best" system settings under all circumstances, they highlight several tradeoffs, in terms of, e.g., the kind of structure, its size, and the computational complexity, that may affect system performance in a non-trivial way. We further make our software tools available to the scientific community to promote sharing of best practices.

1 Introduction

As reported by Cisco [1], the Internet traffic has increased more than fourfold over the past five years. Consequently, the processing speed of network devices such as switches and routers, has grown to let devices process incoming packets at line rate, and pass them to user-space processes for carrying out further analysis – such as intrusion detection, flow management, traffic classification and monitoring, accounting, policying, etc.

Two main trends are keys to this evolution. On one hand, modern Network Interface Cards (NICs) hardware can effectively handle packet rates in the order of tens of Gbps. On the other hand, independent approaches have been proposed to overcome the severe software bottlenecks that affect network stacks of standard operating systems (OS). Examples include PF_RING with Threaded NAPI [18] and variants [13], Netmap [27,28], PacketShader [19] and PFQ [11]. These approaches effectively bypass bottlenecks of standard OS stacks, related to the overhead of per-packet operations like buffer allocation and transfer to user-space, by (i) processing multiple packets in batch to limit IRQs and DMA transactions; (ii) exposing memory of packet buffers to the user-space

A. Dainotti, A. Mahanti, and S. Uhlig (Eds.): TMA 2014, LNCS 8406, pp. 1–14, 2014.
© IFIP International Federation for Information Processing 2014

for zero copy access; (iii) tying every capture thread with its own ring buffer to a fixed CPU to increase cache memory hits (Non-Uniform Memory Access) and (iv) using Receive Side Scaling (RSS) to split incoming flows among different input queues/capture threads. Ultimately, these systems pass packets to user space applications, coping with the worst case of small 64Bytes Ethernet frames at 10Gbps per line card.

As a results of these achievements, bottlenecks have been pushed up to user-level applications, which regardless of their ultimate goal –being it classification, intrusion detection, monitoring, policing, etc.– share a common crucial point. Namely, while low-level hardware and drivers manage *packets*, user-level applications manage *flows*. It follows that a primary, general, concern of user-level applications is to correctly and *efficiently* match packets to the corresponding flow, before taking any subsequent action.

Given that it becomes imperative to perform flow management at line-rate, the goal of this paper is to analyze the design space for flow management, which includes comparing different data structures and hashing functions for keeping a table of flows and updating it by adding, searching and removing flows. By analysing the cost of those operations, we aim at shedding light on tradeoffs when performing and implementing effective flow management modules. Note that here we consider the general case where all packets of a flow have to be matched and processed, which for instance relates to traffic classification and intrusion detection systems; we do not consider the load reduction that packet sampling may have on the flow-matching module, as it potentially benefits only a subset of network applications, e.g., traffic characterization and analysis.

This work builds on top of our previous work [31], in which we implement a multi-threaded statistical "early classification" engine (i.e., based on size of the first few packets of a flow [14]) able to cope with several line cards and to classify *real* traffic at 20 Gbps, or 3.2 Mpps, 116 Kfps (and *synthetic* worst case traffic up to 14.2 Mpps and 2.8 Mfps). Yet, as statistical early classification can be done very efficiently [14, 22], one of the main outcome of [31] was to observe that the flow management module represents the system bottleneck, and that higher traffic processing rates would be possible if this bottleneck was removed. Hence, this paper is motivated by the challenge to understand and overcome the flow management bottleneck, sharing knowledge that can hopefully be useful to a greater extent than the narrow classification focus of [31], to improve performance of generic traffic monitoring and analysis tools.

The rest of the paper is organized as follows. In Sec. 2 we motivate and describe the overall system model. In Sec. 3 we describe our methodology, dataset and workflow, from which we gather results reported in Sec. 4. After comparing our work with related effort in Sec. 5, we summarize the main lesson learned and discuss a number of items in our research agenda in Sec. 6.

2 System Model

The system model we consider in this paper builds over the main lessons learned in [31] concerning flow management, that we describe with the help of Fig. 1. In Fig. 1, our multi-thread system sniff packets, placing them in a flow manager structure, that fires classification operations over batches of packets. A traffic analyzer would have a similar structure, with one or more analyzer modules replacing the classification one. Note that,

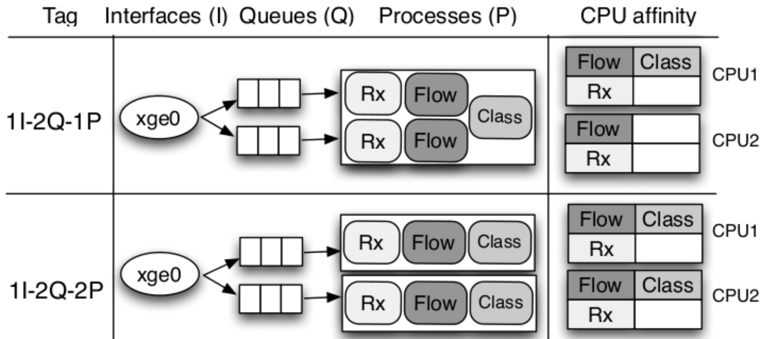

Fig. 1. System model

while the two system configurations in Fig. 1 fire about the same number of threads (represented as shaded block in the rightmost part of Fig. 1, whereas white blocks indicate spare CPU cores), we incurred in severe performance problems (2.1Mpps) whenever two *threads* have concurrently access to the shared flow manager structure (1I-2Q-1P in the figure). Instead, we achieved line rate performance (14.2Mpps) by using two separate *processes*, each of which processed the output of one of the multiple RSS queues exported by the sniffer tools (1I-2Q-2P). Locking issues explain this significant performance gap, and they have to be avoided to allow for sustained system operation [31]. Alternatively, we could avoid locking by replicating the flow manager data structure, letting each thread access a different structure.

Overall, the cost penalty to pay (irrespectively if the solution is mono or multi-process) is a replica of the flow manager data structure to avoid locking. Given a fixed memory budget, it is important to finely tune this structure, as the memory has to be partitioned among multiple independent structures, which are not shared among threads – hence they require no synchronization, or they have no locking issues which reduce the system performance. At the same time, as each structure receives only a portion of the traffic, it may work efficiently with a reduced footprint.

From a more general viewpoint, it would be interesting to optimize flow management operation (as this would possibly lead to extend performance well beyond the 14.2Mpps limit of [31]). Optimization includes not only sizing, but also critical design choices, such as the type of structure (e.g., hash with chaining versus balanced Red-Black trees, etc.) or the choice of the hash function (e.g., CoralReef versus Bob Jenkins, versus Murmur, etc.) that we focus on in this paper.

Otherwise stated, as each process independently receives and analyzes a portion of the whole traffic, we can focus on optimizing an equivalent single-queue single-process system model, as it does not preclude interoperability with recent hardware (e.g., RSS) and software trends early introduced.

3 Methodology

3.1 Trace Driven Emulation

As previously stated, our focus is on optimizing a single pipeline that handles the following operations: (i) fetching a new packet from the queue, which results in loading a new page into the L1 cache, (ii) searching the flow tuple found in the packet into the hash table, which *may* result in either a cache hit or cache miss with page reload.

We hence designed an emulator with an extensible flow tuple feeding interface: to perform the finer grain tuning of the hash table in many significant scenarios we added code for synthetically generating the tuples (e.g., randomly, netscan, portscan, etc.) thus exploring worst case scenarios, and for loading traffic from a trace file. Given that we use backbone traces in the evaluation (whose traffic is notoriously asymmetric), in this work we consider each direction of a traffic flow in isolation for the sake of simplicity. Yet we point out that our results easily extend to bidirectional traffic by XORing source and destination IP addresses and ports before hashing the 5-tuple (which would lead both forward and backward flow directions to be hashed consistently). To evaluate performance of the flow handling code, the emulator loops multiple times over a two step section that stores the tuples into memory first, and then count the time required for matching the tuples into the hash table, eventually adding missing flows, and removing expired ones. While we consider TCP and UDP ports in the flow-matching stage, we do not try to reorder packets by inspecting TCP sequence numbers. As our aim is to focus on general flow-matching performance, we argue that only some applications may require TCP stream reconstruction (e.g., intrusion detection may block subsequent packets of a flow regardless of whether they are received in-order). As such, flow reordering, as for the scope of this work, is left to the spare CPU cores of Fig. 1 and is not performed in what follows.

We release this emulator as an open source software at [2]: with two thousands lines of C code, it embeds both the flow feeding interfaces, the main hash table look-up and the collision managers. About the latter, while the one based on lists pre-allocates directly in the hash table one empty flow element per table line, the one based on the STL RBtree library creates an hash table of tree roots [21]. Clearly this has an impact when chaining is low, as the first flow inside a list is simply copied inside the corresponding empty flow element, while the analogous in the empty tree root requires an initial tree adjustment. Pre-allocating an empty element in the tree without changing the STL library is not possible, that could lead to memory fragmentation issues (see Sec. 6 for future research directions).

While our ultimate goal is to offer insight to fine tune real operational systems such as [31], an experimental approach (i.e., with real traffic over real links) is not suitable for the kind of analysis we carry out in this paper for two main reasons. The first is related to the generality, extensibility and repeatability of our findings. Had we performed tests over the system presented in [31], lessons learned may be of more limited utility for the scientific community. Instead, here we point out that spare CPU cores of Fig. 1 could run any kind of specialized traffic post-process, forwarding, or analysis (whose benchmark is outside the scope of this paper). Second, using the complete network testbed is impractical, and possibly leads to severe bias: indeed, in order to profile each block

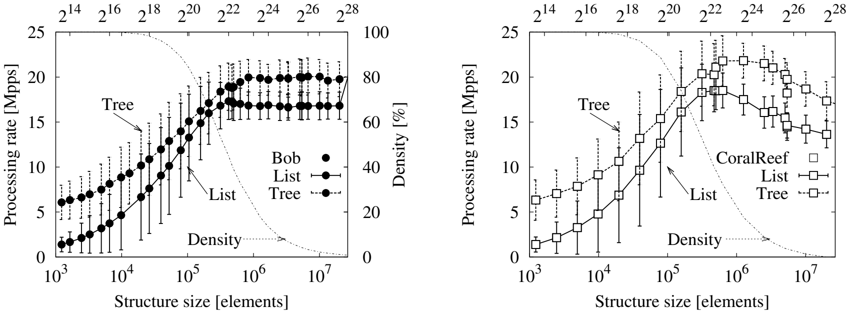

Fig. 2. Structure size tradeoff: Rates of the flow matching process, in Mpps, for different structures and hash functions, as a function of the structure size

to assess the flow matching performance, the system itself was modified introducing a measurement overhead that could impact the performance of the system itself.

As a consequence, in the following we will report our findings only for the trace driven approach. To this purpose, we consider the first 150M packets of the PAIX 2005-01-21 trace from CAIDA [3], collected on a OC48 trunk of an US Commercial Tier1 backbone link connecting San Jose and Seattle.

3.2 Fine Tuning of the Emulation Environment

As a side effect of trace-driven approaches, the data source may run much faster than in the complete experiment, leading to a throughput order of magnitude greater than the original 10Gbps. The reason is two-fold: (i) we store only tuples into memory, packing each packet into a 16 bytes structure, (ii) memory access time is not comparable to the speed of a DMA based system.

While this has no direct impact on the maximum number of flow look up that can be determined *per se*, it makes garbage collection of the hash table critical. If we compute the age of the flows in the hash table using the system clock, the emulation ends up with a greater average occupancy of the table with respect to the network setup (we feed the table with more flows per second while keeping the same time-out age). If otherwise we simulate a time horizon by adding up the delay required to transmit every single packet over a 10Gbps channel (in that case we would have to store each packet length together with the corresponding tuple), we alter the performance evaluation as the flow matching is in real time. For this reason, we opt for using the system clock: while this does not accurately account for performance of the real system, it represents a conservative lower bound of the actual performance (due to higher average occupancy of the data structure with respect to the expected one in real time).

All emulation results are gathered over a 4-cores 3.60 GHz Intel Xeon ES1620 CPU board, equipped with 4x8 GB DIM DDR3 1600Mz RAM memory modules, running Ubuntu 12.04LTS with 64bit Linux kernel version 3.5. As empirical settings of the emulation environment may severely affect the emulation outcomes, we disabled any “smart” feature that dynamically change the rate at which the CPU is working, by (i)

disabling hyper-threading features at the BIOS level (ii) fixing CPU frequency governors to `performance` settings, so that CPUs always run at full 3.60 GHz rate (unlike in default `on-demand` configuration that dynamically tunes the CPU frequency settings).

Similarly, we notice that an aggressive caching policy of the Linux kernel severely affects the performance, depending on the data structure being used: e.g., while in case of lists the whole structure is proactively allocated, in case of trees allocations happen on demand, and system caching may impact pages where the memory is allocated. We therefore sync and drop caches (`/proc/sys/vm/drop_caches`) prior to run any emulation to remove bias due to kernel-level memory management policies.

4 Results

We explored over 850 configurations in terms of hash functions, structures types, size settings, and input traffic. Here we report the most interesting trends that we have observed.

We highlight important tradeoffs concerning settings of (i) the structure size, (ii) the algorithmic complexity of the structure management and (iii) the computational complexity of the hash function. These tradeoffs help architectural decisions, and fine tune the flow management module. However, we acknowledge that some of these aspects (e.g., structure size) depends on the input traffic. Hence the proposed tradeoffs point to two main classes of architectures depending on the network scenario envisioned, rather than to a single candidate solution.

Finally, to extend the validity of our findings, we report a sensitivity analysis in terms of (iv) misconfiguration of the flow manager structure and (v) input traffic.

4.1 Structure Size Tradeoff

In Fig. 2, we report the impact of the structure size on system performance, expressed as the packet processing rate, in packet per second. We consider both list (solid line) and trees (dashed line), with either Bob Jenkins [4] (filled circles, left plot) or CoralReef [5] (empty squares, right plot) hashes, varying the structure size from 2^{14} (16K elements) to 2^{28} (268M elements). Left y-axis reports flow matching rate in Mpps (lines with points and confidence interval), whereas right y-axis reports the density of the structure, i.e., the ratio between the used over the total number of rows in the hash table (dotted line, same for lists and trees).

For very small structure sizes, the depth of the tree or the length of the chain dominate the performance, leading to poor flow matching performance. Performance increase nearly logarithmically for both trees and lists (notice the linear slope but the logarithmic x-axis) up to a certain threshold, close to 2^{22} (4M elements) for the CAIDA trace, whose precise value is related to the spatio-temporal traffic mixture.

Extending the structure size beyond the threshold either does not bring any advantage (Bob, left) or even possibly lead to performance penalties (CoralReef, right). In practice, depending on how the structures have been allocated (proactive in case of hash resolving collision by chaining, or reactive in case of trees) pointers can refer to memory areas

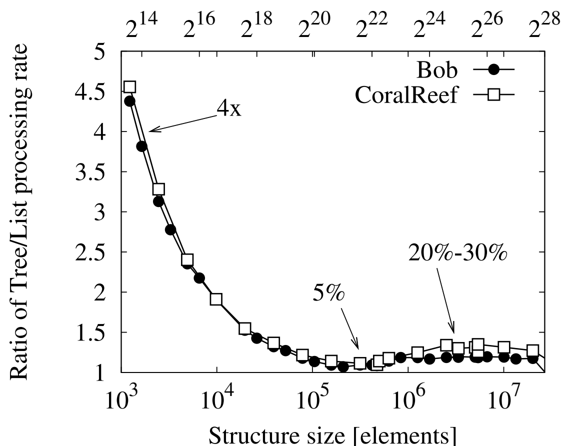

Fig. 3. Structure complexity tradeoff: Ratio between flow matching rates under Trees/Lists

that are stored in different pages, which triggers major page faults and possibly entails quite important performance losses (i.e., about 20% loss in this case for the largest structures).

Otherwise stated, collisions dominate performance of small structures, making the overall throughput low, as expected. Yet, indiscriminately extending the structure size beyond a certain threshold does not necessarily payoff either. Clearly, such over-provisioning situations should be avoided because of bad usage of the allowed memory budget (both Bob and CoralReef cases) and additionally due to possible flow-management performance loss (CoralReef case only). This means that a calibration phase is needed prior to run a tool in a different network environment, or after network upgrade of reconfiguration, and even possibly after traffic mix changes in the same network over longer timeframes.

4.2 Structure Complexity Tradeoff

Fig. 2 also shows another interesting tradeoff that concerns the complexity of algorithmic management. In case of chaining, walking a list of pointers only happens in case of collisions. Conversely, in case of trees more operations have to be done at each element insertion (as the hash points to a memory location containing a pointer toward the root of the tree, implying two memory operations even when the tree has a single element).

Hence, there is an implicit penalty in management of memory pointers in balanced trees, that though often left out of the complexity equation in textbooks, may have an important impact in practice. This can be evinced from Fig. 3 that reports the ratio between flow matching rates under Trees/Lists: it can be gathered that, though a performance improvement exists, it is more significant only in case of improper sizing of the data structures. More particularly: while for very small structures, gain can be up to a factor of 4 and above; for very large structures, gain tops to about than 20-30%; finally,

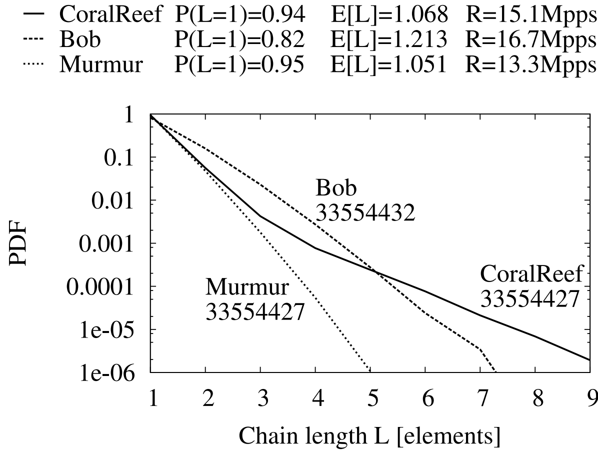

Fig. 4. Computational complexity tradeoff: Chain length probability distribution function (PDF) for CoralReef, Bob and Murmur hash functions

at the critical threshold, gain is about 5%, hinting that proper sizing may play a more important role than structures.

For future work (see Sec. 6), it would be interesting to either consider other data structures (e.g., double hash that avoid chaining), or propose simple tricks to better exploit the simplest data structures (since, as we have seen, performance gain are not necessarily worth the implementation hassle).

4.3 Computational Complexity Tradeoff

Similarly, while different hash functions yield to different amount of collisions, chain length (or, tree depth) explains only part of the story. Indeed, computation of the hash function also consumes resources and impact the flow matching performance.

To highlight this point, we report in Fig. 4 the chain length PDF for the CoralReef, Bob and Murmur hash functions: it can be seen that chains are shorter for Murmur than for Bob or for CoralReef (the latter yielding to longest chains). Yet, notice that the PDF exponentially decreases, which means that the effect of the chain length in case of a properly configured structure size will result in a second order effect.

The figure also reports the probabilities that no chain walking is needed $P(L = 1)$, and the average chain length $E[L]$. Although Murmur hash reduces the collision probability $P(L > 1)$ and average chain length, performance penalty arise in terms of the flow matching rates, due to its implementation complexity[1], which makes it less suitable than simpler CoralReef or Bob hash functions. Hence, in case of Murmur the gain in terms of a lower collision probability and lower chain length are completely offset by the computational complexity needed to achieve it – which makes interesting

[1] We are using the murmur implementation offered by authors `http://code.google.com/p/smhasher/`.

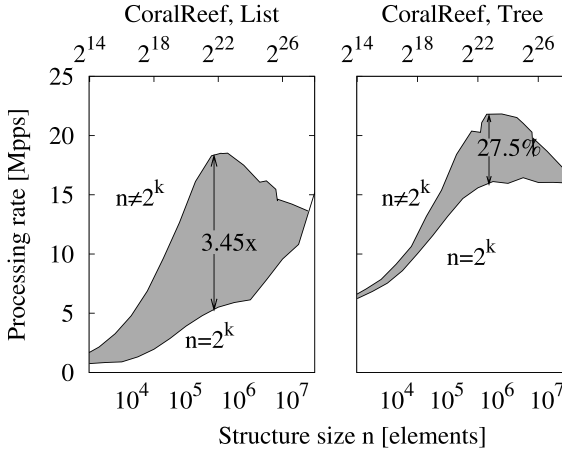

Fig. 5. Misconfiguration of CoralReef structure size

to explore other, faster, alternatives, such as Hsieh hash function [6] or to exploit the pre-computed RSS hash values that are exposed from Intel NICs (see Sec. 6).

4.4 Misconfiguration

As we previously stated, the size of the hash should be tuned according to the workload, as (i) a fixed memory budget must be split among processes running over multiple cores and (ii) performance loss is possible due to unnecessarily large data structures. As such, there is an incentive for researchers and network administrators to finely tune the size of the flow management structure. Problems may raise in case the architecture allows to tune such low level details (e.g., at compile time, or through a configuration file), as careless (or improper) tuning is error prone and can significantly harm the architecture performance.

For instance, while hash functions such as Murmur are inherently robust to the structure size, other hash functions such as CoralReef are not. As we have seen, simpler, computationally faster, hash function such as CoralReef may be preferred in some scenarios. It follows that a tool shipped with a default configuration that is robust for Murmur, may not be robust when the hash function changes. Or, a tool with a default configuration that is robust for CoralReef, may no longer be robust when the structure size is improperly tuned.

Fig. 5 outlines the potential loss for trees and lists. We depict a grey shaded zone between two envelopes: the upper bound is obtained for sizes of the structures that are not powers of two, whereas lower bounds are obtained for powers of two, that are notoriously to avoid in case of CoralReef. The loss in the flow management rate can account to up a factor of 4 in case of lists, and to nearly 30% in case of tree structures. Interestingly, the performance loss is more contained in case of balanced trees, which are inherently more robust to misconfigurations.

Table 1. Bounds to the flow matching rates, CoralReef (54M elements)

	List	Trees
Deterministic	58.1 Mpps	44.6 Mpps
CAIDA	14.6 Mpps	19.7 Mpps
Random	3.1 Mpps	4.6 Mpps

4.5 Input Traffic

While results in the previous section are *qualitative* for general network traffic, *quantitative* performances are bound to the specific trace being used, as well as the hardware capabilities of our emulation environment. Intuitively, flow matching performance can be quantitatively bound between two extreme cases, which are unrealistic as far as the network traffic is concerned: one where the traffic is completely deterministic, vs one where the traffic tuples are completely random.

Interesting observations emerge from Tab. 1. First, as expected real-traffic performance fall in between completely deterministic and completely random 5-tuple sequences. In a sense, this is expected as the entropy of the tuple sequence in the trace is not as high as that of a completely random sequence, since due to temporal scoping of flows, and to heavy hitters, some tuples will be more likely to appear in the sequence. At the same time, skew in the tuple distribution is lower than in the deterministic case, where a unique tuple is constantly hit. As a side comment, whereas trees perform better under heavy stress scenario, as they balance the depth of the structure in case of random or real traffic, in our implementation that requires two memory access their performance is lower in case of deterministic traffic.

Clearly, performance of all possible real traffic sequences fall in between the two ideal extremes presented in Tab. 1. As part of our future work, we plan to investigate the relationship of flow-matching performance with the entropy of the tuple sequence (e.g., a simple way to interpolate between two extremes is to model the probability of individual IP addresses with a Zipf distribution, and tune the skew α). We have tested with multiple traces and artificial sequences. Ideally, we would like to carry out a worst-case analysis with adversarial traffic, though this is unknown and hard to generate in the case of general hash functions. For future work, we aim at injecting DoS or DDoS attacks (e.g., portscan, netscan, etc.) into real traces.

5 Related Work

Monitoring at the flow level requires matching each packet to the correct flow bin. In software-based solutions such as Bro [25], Snort [29], Tstat [30], CoralReef [5] or YAF [20] this is usually accomplished by using hash-based structures over the flow 5-tuple. In [23] the authors point out that most time-consuming operations in systems such as Bro and Snort, are related to tracking the connections – which precisely motivates this work.

Table 2. Maximum Mfps, Mpps and Gbps processing rates of related work

Category	*Ref.*	*Rates*			*Comments*
		Mfps	Mpps	Gbps	
Flow management	[7]	-	6	10	Endace DAG cards
	[8]	-	17	-	16 cores, 16x1Gbps cards
	[15]	1	10	10	6 cores
	[31]	2.8	14.2	10	2 cores

However, to the best of our knowledge, the performance of flow matching code in complex monitoring and intrusion detection systems is rarely evaluated *in a systematic fashion*. In particular, only limited works report on the performance of the data structure and of the hashing functions being used when implementing such operations [24]. Our work indeed starts from similar viewpoint of [24], that however limitedly focus on the study of hash functions, but extends it to consider a more comprehensive set of design choices (e.g., list vs trees, structure size, etc.). Basically, different systems implement their own strategies and we are not aware of work that investigates the impact of such choices in the design of a flow management module. For instance, [20] describes a flow management module in detail, explaining how to optimize flow management using slab allocator [12] for fast recycling of expired flow records, but benchmarks of the system performance are not publicly available. Otherwise, performance analysis for flow matching modules has been done either monitoring real ISP deployments [17] or over offline traces [23, 30, 33]. Comparing the performance of flow management modules of heterogeneous systems (e.g., Bro, Snort, Tstat and YAF) is hard since the *full set* of operations performed beyond flow-management are different, and so are the traces used as input to the evaluation. As such, extrapolating such data from overall measurements [17, 20, 23, 30, 33] can be misleading: to prevent this risk, we rely on publicly available dataset and software.

Explicit performance for systems using dedicated hardware is reported instead in [7, 8, 15, 26, 32], that we summarize in Tab. 2. In [7], using a dual Xeon box hosting a dedicate Endace DAG card, authors match flows at a rate up to 6 Mpps. In [26] an Intel IXP2850 Network Processor is shown matching 10 million concurrent flows at 10 Gbps at full packet rate. Switching to off-the-shelf setup, an application note from Intel [8] reports flow matching of trains of 64 bytes packets at 17 Mpps out of 24 Mpps received over 16$\times$ 1 Gbps interfaces, where each NIC is tied to a different core of an Intel multi-core CPU system (unfortunately the study does not report the number of concurrent flows). A similar architecture [15] matches up to 11Mpps for 1 million concurrent flows at 10Gbps using “FastFlow” algorithms spawned over 6 cores. Instead, the software-based system we proposed in [31], handles aggregate flow rates up to 2.8Mfps using just two cores. Knowledge gathered in this work can hopefully extend further these performance through a fine tuning of the flow management module.

6 Conclusion and Perspective

This work ventures in the flow management component common to all traffic monitors and analyzers. We take a systematic approach, and study the impact of hash

functions, data structure design and sizes in the flow management performance. Employing a trace-driven emulation approach, that allows to jointly gather realistic performance while studying a large design space at the same time, we unveil several tradeoffs in this exploration.

We can summarize our main lessons as:

- *Balanced trees are inherently more robust* to misconfiguration with respect to lists, limiting performance losses.
- *Balanced trees are inherently more complex to manage*, to the point that frequent memory operations may erode the advantage over simples structures such as lists, where infrequent collisions in case of properly configured structure sizes, translate into fewer memory operations.
- *Hash functions play a minor role,* at least when structures are properly sized, with computational complexity eroding the advantage of hashes with better entropy properties.

Clearly, this work is by no means complete: expansions can include a larger spectrum of *hash functions* (e.g., CRC32, One-at-a-Time, FNV [9], Hsieh [6] among others, especially aiming at lower computational complexity) or *data structures* (e.g., double hash, cuckoo hashing [34], denser hashes as in DPDK [10]).

Additionally, low-level system aspects such as memory management, including translation lookaside buffer, memory fragmentation and alignment, and NUMA allocation (the latter considered by us in [31]), likely play an important role and deserve attention.

A second direction is to replicate this study over a *wider dataset* including traces from different network segments, in an attempt to find consistently good setting that can be recommended for different environments. A useful extension of this work would be to correlate flow-management performance (e.g., matching rate, chain length, tree depth, etc.) with *key characteristics* of the network traffic (e.g., distributions of address space, spatio-temporal correlation of arrivals, etc.) to also offer a methodology for a semi-automatic fine-grained tuning of the above data structures.

Security is another interesting topic that, due to space constraints, was left out of the scope of this paper (see [16] and references therein). For future work, we aim at injecting DoS or DDoS attacks (e.g., portscan, netscan, etc.) into real traces, and to investigate adversarial scenarios (leading to hash table collision). Such adversarial *synthetic patterns* could be super-imposed over real traces: as due their distributed nature, with possibly spoofed addresses, attacks could be used to stress-test the flow management architectures. Additionally, this would also allow to tune the level of randomness between real vs random traffic, by specifying the intensity of the synthetic pattern with respect to the normal traffic.

Acknowledgement. This work has been carried out at LINCS `http://www.lincs.fr`. The research leading to these results has received funding from the European Union under the FP7 Grant Agreement n. 318627 (Integrated Project "mPlane").

References

1. http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-481360.pdf
2. http://www.ing.unibs.it/ntw/tools/fmsim
3. http://www.caida.org/data/passive/trace_stats/
4. http://burtleburtle.net/bob/
5. http://www.caida.org/tools/measurement/coralreef/dists/coral-3.9.1.tar.gz
6. http://www.azillionmonkeys.com/qed/hash.html
7. http://www.terena.org/activities/ngn-ws/ws2/deri-10g.pdf
8. http://download.intel.com/design/intarch/papers/322516.pdf
9. http://www.isthe.com/chongo/tech/comp/fnv/
10. http://www.dpdk.org
11. Bonelli, N., Di Pietro, A., Giordano, S., Procissi, G.: On multi–gigabit packet capturing with multi–core commodity hardware. In: Taft, N., Ricciato, F. (eds.) PAM 2012. LNCS, vol. 7192, pp. 64–73. Springer, Heidelberg (2012)
12. Bonwick, J.: The slab allocator: An object-caching kernel memory allocator. In: USENIX Summer Technical Conference (1994)
13. Cardigliano, A., Deri, L., Gasparakis, J., Fusco, F.: vPF_RING: Towards wire-speed network monitoring using virtual machines. In: ACM IMC (2011)
14. Crotti, M., Dusi, M., Gringoli, F., Salgarelli, L.: Traffic classification through simple statistical fingerprinting. ACM SIGCOMM Comput. Commun. Rev. 37(1), 5–16 (2007)
15. Danelutto, M., Deri, L., De Sensi, D.: Network monitoring on multicores with algorithmic skeletons. In: International Conference on Parallel Computing, PARCO (2011)
16. Eckhoff, D., Limmer, T., Dressler, F.: Hash tables for efficient flow monitoring: vulnerabilities and countermeasures. In: IEEE LCN (2009)
17. Finamore, A., Mellia, M., Meo, M., Munafo, M., Rossi, D.: Experiences of Internet traffic monitoring with Tstat. IEEE Network 25(3), 8–14 (2011)
18. Fusco, F., Deri, L.: High speed network traffic analysis with commodity multi-core systems. In: ACM IMC (2010)
19. Han, S., Jang, K., Park, K., Moon, S.: PacketShader: a GPU-accelerated software router. In: ACM SIGCOMM (2010)
20. Inacio, C., Trammell, B.: YAF: yet another flowmeter. In: International Conference on Large Installation System Administration, LISA (2010)
21. Knuth, D.E.: The art of computer programming (1968)
22. Lim, Y., Kim, H., Jeong, J., Kim, C., Kwon, T., Choi, Y.: Internet traffic classification demystified: on the sources of the discriminative power. In: ACM CoNEXT (2010)
23. Lin, P.-C., Lee, J.-H.: Re-examining the performance bottleneck in a nids with detailed profiling. Journal of Network and Computer Applications 36(2), 768–780 (2013)
24. Molina, M., Niccolini, S., Duffield, N.: A comparative experimental study of hash functions applied to packet sampling. In: International Teletraffic Congress, ITC (2005)
25. Paxson, V.: Bro: a system for detecting network intruders in real-time. Computer Networks 31(23-24), 2435–2463 (1999)
26. Qi, Y., Xu, B., He, F., Yang, B., Yu, J., Li, J.: Towards high-performance flow-level packet processing on multi-core network processors. In: ACM/IEEE ANCS (2007)
27. Rizzo, L.: Netmap: a novel framework for fast packet I/O. In: USENIX Annual Technical Conference (2012)
28. Rizzo, L., Carbone, M., Catalli, G.: Transparent acceleration of software packet forwarding using netmap. In: IEEE INFOCOM (2012)

29. Roesch, M.: Snort - lightweight intrusion detection for networks. In: USENIX Conference on System Administration (1999)
30. Rossi, D., Mellia, M.: Real-time TCP/IP analysis with common hardware. In: IEEE ICC (2006)
31. Santiago del Río, P.M., Rossi, D., Gringoli, F., Nava, L., Salgarelli, L., Aracil, J.: Wire-speed statistical classification of network traffic on commodity hardware. In: ACM IMC (2012)
32. Srinivasan, D., Feng, W.: Performance analysis of multi-dimensional packet classification on programmable network processors. Computer Communications 28(15), 1752–1760 (2005)
33. Wang, D., Xue, Y., Dong, Y.: Memory-efficient hypercube flow table for packet processing on multi-cores. In: IEEE GLOBECOM (2011)
34. Zhou, D., Fan, B., Lim, H., Kaminsky, M., Andersen, D.G.: Scalable, high performance ethernet forwarding with cuckooswitch. In: ACM CoNEXT (2013)