On the Design of Efficient Caching Systems

Lorenzo Saino

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
of
University College London.

Department of Electronic and Electrical Engineering
University College London

2015

I, Lorenzo Saino, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the thesis.

Department of Electronic and Electrical Engineering
University College London

Abstract

Content distribution is currently the prevalent Internet use case, accounting for the majority of global Internet traffic and growing exponentially. There is general consensus that the most effective method to deal with the large amount of content demand is through the deployment of massively distributed caching infrastructures as the means to localise content delivery traffic. Solutions based on caching have been already widely deployed through Content Delivery Networks. Ubiquitous caching is also a fundamental aspect of the emerging Information-Centric Networking paradigm which aims to rethink the current Internet architecture for long term evolution. Distributed content caching systems are expected to grow substantially in the future, in terms of both footprint and traffic carried and, as such, will become substantially more complex and costly.

This thesis addresses the problem of designing scalable and cost-effective distributed caching systems that will be able to efficiently support the expected massive growth of content traffic and makes three distinct contributions. First, it produces an extensive theoretical characterisation of sharding, which is a widely used technique to allocate data items to resources of a distributed system according to a hash function. Based on the findings unveiled by this analysis, two systems are designed contributing to the abovementioned objective. The first is a framework and related algorithms for enabling efficient load-balanced content caching. This solution provides qualitative advantages over previously proposed solutions, such as ease of modelling and availability of knobs to fine-tune performance, as well as quantitative advantages, such as 2x increase in cache hit ratio and 19-33% reduction in load imbalance while maintaining comparable latency to other approaches. The second is the design and implementation of a caching node enabling 20 Gbps speeds based on inexpensive commodity hardware. We believe these contributions advance significantly the state of the art in distributed caching systems.

Acknowledgements

The work presented in this thesis would have not been possible without the support, advice and encouragement received from a multitude of people to whom I would like to express my most sincere gratitude.

First and foremost, I would like to thank my supervisor, George Pavlou, for giving me the opportunity to pursue this research and for constantly providing me with the encouragement, advice, freedom and funding that made this work possible.

I am also very grateful to Ioannis Psaras for countless discussions and insightful advice that guided my work and helped shaping this thesis.

I benefited from the advice provided by many mentors that I encountered along the way. They include all current and former members of the Networks and Services Research Lab at UCL, my advisors at Bell Labs, Diego Perino and Massimo Gallo, and my thesis examiners, George Polyzos and Miguel Rodrigues.

I also received a great amount of encouragement from many friends with whom I had the pleasure to share my time in London, Paris and my hometown and in particular from Meihong Zheng, who greatly supported me over the last few months.

Lastly, and most importantly, I am deeply grateful to my family, in particular my father Ermano, my sister Laura and Claudia. They offered me constant encouragement and support throughout all my education and have been a major source of inspiration for all my achievements.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the last decade, the amount of Internet traffic accounting for content distribution has increased dramatically. As an example, in 2014, video alone accounted for 67% of the global Internet traffic. This figure is forecast to steadily increase up to reaching a share of 80% of global Internet traffic by 2019 [177].

This staggering increase in content delivery traffic led to an intensification of a number of associated phenomena challenging the stability and the reliability of the Internet infrastructure. These include for example flash crowd events, which consist in a large number of users suddenly and unexpectedly requesting in large volume one or few content items from a specific content provider. Such events have become extremely common and have the potential to heavily impact the operations of the affected content providers if appropriate countermeasures or proactive capacity planning are not put in place [180]. A recent example of the magnitude at which flash crowds can shift traffic across the Internet occurred on April 12, 2015, day of the season five premiere of Game of Thrones. On that day, the amount of traffic generated by HBO's two streaming services (HBOGO and HBONOW) accounted for 4.1% of traffic on one US fixed network, representing an increase of over 300% of their average levels [151].

Flash crowd events can also take place as a result of malicious activity, such as Distributed Denial of Service (DDoS) attacks. A recent instance of these attacks occurred in March 2015, when the website `greatfire.org` (an organisation contrasting Internet censorship in China) was flooded with a large amount of traffic, with the objective of disrupting its operations. The attack was perpetrated through a man-in-the-middle attack orchestrated by an infrastructure co-located with China's Great Firewall that became known as the Chinese "Great cannon" infrastructure [124].

This rapid and constant increase in content distribution traffic and related phenomena have been severely challenging the scalability of the global Internet. This adverse impact is partly due to the fact that the Internet was originally designed envisaging host-to-host communications as its main use case and, as such, it is ill-suited for content distribution, which is now its main use case.

There is general consensus that the most effective solution to content distribution problems is through the adoption of globally distributed in-network caches. This has led, as a first short-term solution, to the widespread deployment of Content Delivery Networks (CDNs), which have rapidly grown in terms of both presence and overall traffic carried. This trend is forecast to continue steadily over the coming years

to the point that, by 2019, 62% of Internet traffic will cross a CDN [177]. This will also lead to a massive growth of the CDN market, which is estimated to reach the size of 15.73 billion USD by 2020 [125].

Other solutions have also been proposed to address the problem more systematically in the long term, by realigning Internet architecture to its prevalent content distribution use case. For example, the recently proposed Information-Centric Networking (ICN) paradigm aims to shift away from the current Internet architecture, where the main abstraction is represented by location-dependent node addresses and transition to an architecture where the main abstraction will be location-agnostic content identifiers. Many specific architectures have been proposed to implement this paradigm. The two most prominent ones, Content-Centric Networking (CCN) [89] and Named Data Networking (NDN) [183] both envisage the ubiquitous deployment of content caches on every network router.

The use of distributed caching is beneficial not only as a defensive measure for dealing with rapidly increasing traffic but also as a tool for improving user experience by reducing latency. In fact, geographically distributing replicas of the most requested content, makes it possible for users to retrieve content from closer locations and hence experiencing lower latency and greater throughput. Latency is extremely important for user experience as demonstrated by many experiments. For example, Google showed that a 500 ms latency increase in showing results from a Web search (caused by showing 30 results instead of 10) resulted in a 20% drop in searches. Similarly, Amazon estimates that a latency increase of 100 ms causes a 1% drop in sales [107].

As the footprint and traffic carried by these networks of caches increases so does their complexity and cost. It is therefore of crucial importance to build scalable and efficient networked caching systems that can scale well to support rapidly increasing traffic and are cost-effective. There are several areas where improvements can be made, from the design of single caching nodes to the planning, design and operation of a distributed system of such nodes. This thesis provides analytical, architectural and design contributions touching upon both single caching nodes and their interaction in a cooperative cache network.

1.1 Problem statement

Content caching is generally recognised as the most appropriate countermeasure to deal with the exponential growth in content traffic as discussed above. However, deploying and operating large-scale distributed caching systems that are capable of reliably handling a large amount of traffic is both complex and costly. For this reason, a key challenge to ensure that caching remains an effective solution despite the exponential increase in content traffic is to improve the *efficiency* of caching systems, and more specifically to address their *scalablity* and *cost-effectiveness*.

Therefore, this thesis attempts to answer the following question:

How can content caching systems be made more efficient, scalable and cost-effective?

The focus of this thesis on *efficiency*, *scalability* and *cost-effectiveness* is mostly motivated by the necessity to counteract the constant increase in content traffic that impacts the global Internet with techniques that are directly applicable in practice.

1.2 Contributions

This thesis makes the following contributions.

Modelling of load-balancing and cache performance properties of sharded systems. Sharding is a method for allocating data items to nodes of a distributed caching or storage system based on the output of a hash function computed on the item identifier. It is ubiquitously used in key-value stores, CDNs and many other applications. Despite considerable work has focused on the design and the implementation of such systems, there is still limited understanding of their performance in realistic operational conditions from a theoretical standpoint. We fill this gap by providing a thorough modelling of sharded caching systems, focusing particularly on load balancing and caching performance aspects. This analysis provides important insights that can be applied to optimise the design and configuration of sharded caching systems.

Design of a framework and algorithms for load-balanced content caching. We design a complete framework that aims to achieve load-balanced content caching in ISP-controlled network caches. Our approach applies sharding techniques to the case of geographically distributed caches. The resulting framework has several qualitative advantages including performance predictability and availability of knobs enabling fine-tuning of performance. This solution achieves on average a 2x cache hit ratio increase and a 19-33% reduction in load imbalance across links in comparison to previously proposed techniques, while still achieving comparable latency.

Design and implementation of a caching node for content distribution. Read-through caching nodes are a fundamental building block of content delivery infrastructures. We design and implement a high-speed cost-effective caching node that is able to operate at line speed on inexpensive commodity hardware, thanks to intelligent software optimisations. We show with trace-driven experiments that our prototype is able to serve 20 Gbps of traffic even with modest hardware.

In addition to the contributions listed above, this thesis also includes the following side contributions that are related to the methodology used to evaluate the primary contributions.

Design and implementation of a framework for deploying network experiments. Arguably, one of the most cumbersome tasks required to run a controlled network experiment is the setup of a complete scenario and its implementation in the target simulator or emulator. We tackle this problem by designing and implementing a toolchain allowing users to easily create a network experiment scenario and deploy it on a number of simulators or emulators. The automation provided by this tool strongly simplifies the creation of a network scenario, which would otherwise be both time-consuming and error-prone.

Design and implementation of a scalable and extensible caching simulator. Evaluating the performance of complex networked caching systems reliably and efficiently requires a scalable simulation platform that is able to process a large number of requests in reasonable time. This is because a large number of requests is required to reach the steady state of a network of caches. Such a

simulator must also be easily extensible to simplify the implementation and evaluation of new models and algorithms. To address all these requirements, we design and implement a highly scalable and extensible caching simulator. The abstractions and optimisations used in our design provide extremely accurate results but with considerably less processing and memory overhead than other simulators. The abstractions we devised in our tool have been later adopted by other simulators.

1.3 Thesis outline

This thesis is organised as follows.

Chapter 2 provides an overview of distributed caching and how it can be used to improve the efficiency of content distribution by surveying use cases, problem space and previous work.

Chapter 3 presents a thorough analytical modelling that sheds light on key properties of sharded systems.

Chapter 4 builds upon results of Chapter 3 and shows how sharding techniques can be applied to build a load-balanced, robust and scalable ISP-wide caching system.

Chapter 5 also builds upon results of Chapter 3 and presents the design and implementation of a cost-effective high-speed caching node.

Chapter 6 draws conclusions on the present work and discusses potential directions for future work.

Appendix A presents a methodology for simplifying the setup of a controlled network experiments and describes the design and implementation of *FNSS*, a toolchain implementing it.

Appendix B presents the design and implementation of *Icarus*, a scalable and extensible caching simulator which has been used for producing the results presented in chapters 3 and 4.

Chapter 2

In-network caching: state of the art

The contributions of this thesis concern the analysis, design and implementation of networked caching systems. The general problem of data caching has been widely investigated before in a variety of computer systems applications such as CPUs, storage systems and databases. This thesis focuses specifically on networked caching systems for content distribution applications. In particular we are interested in two key applications: current Content Delivery Networks (CDNs) and emerging Information-Centric Networking (ICN) architectures. The objective of this chapter is to provide the necessary background for the original contributions discussed in subsequent chapters of this thesis.

The remainder of this chapter is organised as follows. In Sec. 2.1 we overview CDN and ICN architectures, which are the two main applications targeted by the contributions of this thesis. In Sec. 2.2 we analyse the technical challenges concerning in-network caching and investigate the related design space. In Sec. 2.3, we further delve into the design space by focusing on two key classes of algorithms adopted in networked caching systems, precisely cache replacement algorithms and caching meta algorithms. In Sec. 2.4 we review modelling techniques for caches analysed both in isolation and as part of a network. We conclude by summarising the content of this chapter in Sec. 2.5.

2.1 Applications

2.1.1 Content Delivery Networks

Content Delivery Networks (CDN) are networks of globally distributed caches operating as intermediaries between content providers and users. Their role is to store popular contents belonging to content providers and serve them to users on their behalf.

Given their geographically distributed nature, CDNs can distribute contents more efficiently than the centralised infrastructures of content providers for a number of reasons. First of all, they reduce user-perceived latency, since contents are served from a repository topologically and geographically closer to the end user. The increased proximity to the user also improves throughput. In fact, since the TCP congestion window increase rate is inversely proportional to the Round Trip Time (RTT) of a flow, it is well known that flows with shorter RTT experience a faster congestion window increase and therefore greater throughput. Moreover, the shorter network path also reduces the probability of encountering bottlenecks.

In addition to the improved performance, CDNs also provide tangible economic benefits. In fact, since CDNs normally carry traffic belonging to many content providers, they can take advantage of better economies of scale and therefore deliver traffic more cost-effectively than content providers. As a result, outsourcing content distribution to a CDN may be less expensive than distributing content to end users directly.

CDNs can be categorised in three groups based on their relation to content providers and network operators.

Traditional commercial CDNs. These CDNs are independent entities providing services to content providers. They normally deploy caches at various locations in the world, at Internet Exchange Points (IXPs) and at PoPs of ISPs with whom agreements have been made. Most CDNs fall in this category. Examples include Akamai [134], Velocix [175], Fastly [23] and Limelight [121].

Private CDNs. For very large content providers it is normally more cost-effective to operate their own CDN infrastructure rather than relying on third-party services. Such CDN infrastructures, owned and operated by content providers for their exclusive use, are normally referred to as Private CDNs. Content providers operating a private CDN include Google [72], Facebook [82] and NetFlix [67].

Network/telco CDNs. These are CDNs owned and controlled by a network operator providing content delivery services to third party content providers. Such CDNs have recently grown at a considerable pace, driven by both technological and commercial incentives. Examples of Network CDNs are EdgeCast (recently acquired by Verizon) [53], AT&T [13] and Level3 [117]. Differently from other types of CDNs, Network CDNs have control of both the caching infrastructure and the underlying network infrastructure used to deliver the content. Therefore they can exert more control on network configuration and co-optimise the operations of network and caching infrastructures in a way that is not possible for traditional CDNs.

From a technical standpoint, a CDN is a collection of distributed HTTP reverse proxies, possibly managed by a (logically centralised) control infrastructure. Users request contents by issuing HTTP GET requests to content provider URLs. If the target content provider wishes the request to be served by the CDN, it redirects the request to the CDN through either DNS redirection [38] or IP-layer anycast [41]. In the case of DNS redirection, the content provider redirects a user request to a CDN-managed hostname. This triggers the client to resolve the CDN hostname to an IP address. The authoritative DNS server for the CDN domain then resolves the request to the IP address of a caching node relatively close to the user. If IP-layer anycast is used instead, the content provider directly redirects requests to an anycast address of the CDN. In practice, using IP-layer anycast bypasses the DNS lookup for resolving the CDN hostname but reduces the room for complex server selection.

2.1.2 Information-Centric Networking

As already discussed above, the mismatch between the original Internet design assumptions (host-to-host communications) and current usage patterns (content distribution) has partially been addressed through application-layer solutions, *i.e.,* Content Delivery Networks (CDN), which have retrofitted some desirable

content-aware functionalities on top of the existing architecture. However, the lack of native network support for content distribution restricts the efficiency of such approaches, and also potentially hinders the evolution of the Internet as a whole.

For all these reasons, over the last few years, a considerable research effort has been put into rethinking Internet architecture having in mind user-to-content traffic as the prevalent usage pattern. As a result, a number of Information-Centric Networking (ICN) architectures have been proposed. Examples include CCN [89], NDN [183], DONA [108], COMET [34], PSIRP/PURSUIT [64] and SAIL/4WARD/NetInf [47].

Although the various architectures proposed so far have different objectives and hence exhibit different operational properties, they all share the following common features.

Request-response model. Content items are transferred in a receiver-driven manner. Users issue requests for explicitly named content items that are routed by the network to suitable entities, which then reply with the requested content items.

Location independence. Content names do not reflect the location (host) where the content is stored. Rather, content naming can be based on any, possibly human-readable, sequence of bytes. Names, however, in most ICN architectures are used to route requests to contents and contents to users, hence, naming has to follow a specific pattern depending on the *route by name* approach adopted.

Content-oriented security model. Authenticity of delivered contents is no longer implemented by authenticating the content source and securing the channel between sender and receiver. Instead the authenticity of a content object can be validated regardless of what node actually served the content.

Ubiquitous in-network caching. Explicitly named content objects can be cached in arbitrary locations of the network and be retrieved from there once subsequently requested.

Among the various architectures implementing the ICN paradigm proposed so far, the two architectures which have gained most momentum are CCN [89] and NDN [183]. Both proposals share the same high level design. Their differences are limited to marginal aspects such as packet format and various algorithms that are not of interest for our discussion.

In CCN/NDN there are two types of packets: *Interest* and *Data*. Interest packets are issued by users to request a packet-sized chunk of data. Data packets are sent in response to an Interest packet containing the piece of data requested. All operations in CCN/NDN are receiver-driven, *i.e.,* a content origin cannot proactively push Data packets over the network without being first requested by a receiver through an Interest packet.

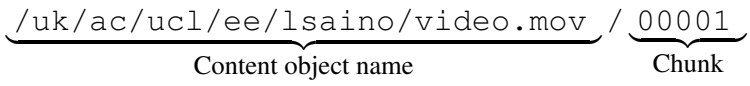


Figure 2.1: CCN/NDN content name example

Content chunks are assigned human-readable hierarchical names. As depicted in Fig. 2.1, each content name comprises two parts: a content object name, which identifies the content object (*i.e.,* file) to

which the chunk belongs and a chunk identifier which is practically the index of the chunk within the content object.

Interest packets are routed towards a close replica of the requested content based on the name of the requested chunk, as opposed to node identifiers as in the current IP-based architecture. Each router therefore needs to implement a name-based forwarding table that is able to perform Longest Prefix Match (LPM) based on content name. Differently, Data packets are not forwarded back to the requester based on some identifiers. Instead, each router keeps track of the interfaces from which it received Interest packets and, when it receives a matching Data packet, it forwards it back over the interfaces from which Interest packets were received based on that information. This mechanism ensures symmetrical reverse-path forwarding.

Each CCN/NDN router comprises three main data structures:

Forwarding Information Base (FIB). It stores next hop routing information for each content prefix and is required for the forwarding of Interest packets.

Pending Interest Table (PIT). It temporally stores the list of interfaces from which Interest packets have been received. It is required to forward Data packets to all routers which issued a matching Interest packet.

Content Store (CS). It caches received Data packets so that future requests for the same chunk can be served by the router.

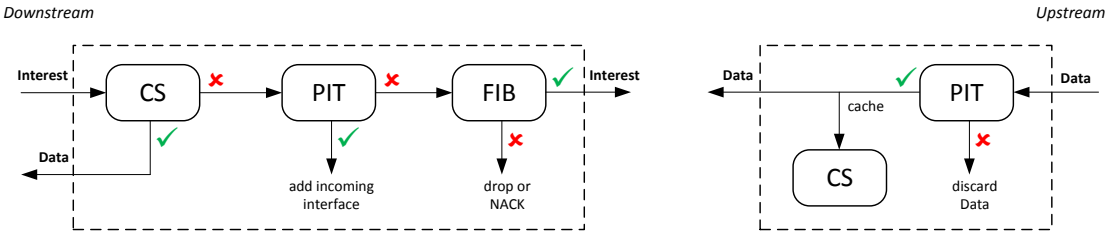


Figure 2.2: CCN/NDN router processing flow (reproduced from [183])

More specifically, a CCN/NDN router operates as follows (see Fig. 2.2). When an Interest packet is received, the CS is looked up: if the requested chunk is present, it is returned to the user. If the CS lookup is unsuccessful, the PIT is looked up next: if there is already an entry for the requested chunk, the interface from which the Interest was received is added to the entry and the Interest is discarded. If also the PIT lookup fails, the FIB is looked up and, if the FIB contains a matching entry, the Interest packet is forwarded to the next hop, otherwise the packet is dropped or possibly flooded over all interfaces. Upon reception of a Data packet, the PIT is looked up first. If a matching entry is found, the Data is forwarded to all interfaces from which an Interest packet was received, the PIT entry is dropped and the Data is cached in the CS. Otherwise, the Data packet is discarded.

In Chapter 5 we describe the design and the implementation of a complete CCN/NDN router comprising a two-layer DRAM/SSD Content Store.

2.2 Problem and design space

In this section, we examine the distributed content caching problem and review the main techniques proposed so far, pointing out their shortcomings that the contributions of this thesis address.

According to [110] the in-network caching problem can be partitioned into three well-defined subproblems:

Content placement and content-to-cache distribution, which deals with the problem of deciding which content items to place in which caching node and how to distribute them to those nodes.

Request-to-cache routing, which deals with how to route content requests from a requester to a suitable caching node that holds a copy of the requested content.

Cache allocation, which deals with optimising placement and sizing of caching nodes.

Following this classification, we now discuss the state of the art in in-network caching, specifically focusing on techniques applicable to the case of operator-controlled caches, which is the use-case investigated in Chapter 4.

2.2.1 Content placement

Generally speaking, content items can be placed in in-network content caches in either a *proactive* or a *reactive* manner.

With proactive content placement, caches are pre-populated during off-peak traffic periods. The placement is normally computed by an off-line optimisation algorithm on the basis of historical data and/or future predictions and repeated periodically, normally every 24 hours. Many algorithms have been proposed to determine optimised content placement under a variety of objective functions and constraints [8], [16], [19], [24], [153].

Instead, under reactive content placement schemes, content items are stored in caches as a result of cache misses in a read-through manner. Within a caching node, content items can be replaced according to a variety of policies, which we survey in Sec. 2.3.1.

In case a request traverses more than one cache before hitting the content, the most simple content placement strategy is to leave a copy of the content in every node traversed, which is known as Leave Copy Everywhere (LCE). However, this strategy causes a high degree of redundancy as all caches along the delivery path consume cache resources to hold identical items.

To reduce this redundancy and, therefore, to increase the number of distinct content items cached along a delivery path, a number of meta algorithms have been proposed to selectively cache each delivered content in a subset of the nodes of the path. These include Leave Copy Down (LCD) [113], "Cache less for more" [33] and ProbCache [139]. We refer the reader to Sec. 2.3.2 for further details.

Proactive content placement allows for a simpler implementation of caching nodes and enables them to achieve better throughput. In fact, since with proactive placement the population of caches occurs during off-peak periods, busy-hour workloads are read-only. This enables multi-core implementations of caches to operate in a lockless manner as no writes occur. In addition, this ensures greater read throughput

from storage technologies such as SSD and HDD, which would provide a reduced read throughput if concurrent writes were executed. We delve into the implementation challenges of reactive caching nodes in Chapter 5.

However, the simpler node implementation offered by proactive placement comes at the cost of two substantial disadvantages. First, it causes rigidity in addressing changes in traffic demand, as any unexpected variation in demand patterns would lead to a reduction in cache hit ratio until a new proactive content placement takes place. Second, the computation of the optimal content placement requires data from both the cache operator (cache network topology, processing capabilities, cache sizes) and the content provider (historic demand and demand predictions), which may be difficult to collect if the cache operator and content provider are different entities.

With regard to cache efficiency, there is consensus that proactive placement is preferable to reactive placement only in the case of specific workloads characterised by a limited content catalogue and predictable request variations, such as Video on Demand (VoD) [8], [42] and adult video content [170]. As a matter of fact, Netflix, the world largest VoD content provider, uses proactive content placement in their video caching infrastructure [67].

Other types of traffic are generally characterised by rapid variations of content popularity that would obliterate the gains provided by an optimised proactive placement. Sharma *et al.* [153] showed that, with respect to Web traffic, placing contents proactively every 24 hours based on previous-day historical data achieves considerably worse performance than using a simple reactive placement. They also showed that even proactively placing content items with exact knowledge of future demand would only provide up to 1-18% performance gain in comparison to reactive placement. As a matter of fact, to the best of our knowledge, all commercial CDNs for Web traffic populate their caches reactively, whether they are designed specifically for static content (*e.g.,* Akamai [122], [134]) or dynamic content (*e.g.,* Fastly [23]). Reactive content placement is also used by dedicated caching infrastructures of large-scale content providers, such as the Facebook photo storage [82] and the Google Global Cache [72]. This is also the placement strategy selected by all ICN architectures that propose ubiquitous packet caches in network routers [89], [183].

In line with the above facts, the design of the distributed caching system of Chapter 4 and the caching node of Chapter 5 both assume reactive caching operations.

2.2.2 Request routing

Request routing strategies can be broadly ascribed into two categories: *opportunistic on-path* and *coordinated off-path* routing.

With on-path request routing, content requests are first routed from the requester to the closest cache. Then, they are routed over the network of caches towards content origin using shortest path routing and are served from a cache only if the requested content item is available at a node on the request path. This routing strategy is highly scalable because it does not require any coordination among caching nodes and can be used in conjunction with either proactive [96], [118] or reactive [33], [113], [139] content placement. However, it may suffer reduced cache hits, especially in dense cache deployments, because

contents cached nearby the requester but not on the shortest path to the origin will never be hit.

It should be noted that *edge caching* is also a (simpler) case of opportunistic on-path routing. In edge caching, requests are routed to the closest cache but in case of a cache miss they are forwarded directly to the content origin. This is for example how Google Global Cache operates [72], where each cache deployed within an ISP network is statically mapped to a subset of requesters and, in case of a cache miss, requests are routed outside the ISP network.

Differently, with off-path coordinated routing, requests can be served by a nearby node even if not on the shortest path to the origin. This however comes at the cost of higher coordination to exchange content availability information among caches.

Off-path routing can be implemented using a centralised or distributed content-to-cache resolution process. In a centralised resolution process, a (logically) centralised entity with a global view of cached contents is queried before routing a content request and it returns the address of the closest node storing the requested content item. This approach is however suitable only for systems operating under proactive content placement or even reactive placement as long as content location varies infrequently. For reactive caching systems with high rate of content replacement (which also include ICN architectures, where items are cached at a chunk-level granularity), a number of more scalable off-path request routing algorithms have been proposed. The main objective is to enable caching nodes to exchange state among each other in a lightweight manner and route requests accordingly. In Breadcrumbs [142], a caching node that forwards a content item records the address of the node towards which it forwarded the item and opportunistically routes future requests for the same content towards the node the content item was forwarded earlier. Another recent approach by Rossini and Rossi [147], approximates an ideal Nearest Replica Routing (NRR) by broadcasting either content requests or meta-requests to discover the location of the closest content item.

Request routing design presents a clear tradeoff between scalability and efficiency. The limited scalability of off-path routing schemes particularly limits the availability of design choices for reactive caching systems and ICN architectures, which are of our interest. In Chapter 4 we present the design of a caching framework that effectively addresses this tradeoff by enabling off-path request routing without any coordination.

2.2.3 Cache allocation

Optimal cache placement and sizing is strictly dependent on content placement, request routing, network topology and request patterns.

Several studies have investigated the optimal cache allocation for the case of proactive content placement, focusing on different operational conditions and objective functions, with good results. See, for example [115] and [116].

However, the problem is considerably more complex in the case of reactive content placement. Simple and practical cache replacement policies, such as LRU and FIFO, are difficult to model, even in a single cache scenario [35], [46], [71]. The main source of complexity comes from modelling the behaviour of caches receiving miss streams of other caches [92]. In fact, although there exist analytical

models capturing well the behaviour of common cache replacement policies in a single cache or tandem configurations [35], [113], [126], extending this analysis to arbitrary network topologies is very hard and methods proposed so far are computationally very expensive and fairly inaccurate [143]. All this makes the optimisation problem hard to solve.

In the specific case of edge caching, since each request traverses a single cache, the problem is widely simplified. In this situation, the optimal cache placement can be mapped to a p-median location-allocation problem [109]. Although finding the optimal solution for the general case is NP-hard [101], there exist algorithms for solving it in polynomial time, optimally for simple topologies [109] or approximately for arbitrary topologies using well-known heuristics [11], [162].

To the best of our knowledge, only two works have investigated the optimal cache placement and sizing in the case of reactive content placement in arbitrary network topologies, although both focus on the specific case of on-path request routing with LCE+LRU content placement. In [146], Rossini and Rossi investigate the performance of heuristics assigning cache capacity to nodes proportionally to certain centrality metrics. They concluded that these simple optimisations do not bring considerable advantages though. In [178], Wang *et al.* formulate the optimal cache placement problem as a standard knapsack problem with constrained cumulative cache size. Despite their solution is not provably optimal, they reported that for the scenarios investigated, this cache placement strategy improved upon the heuristics proposed in [146].

The caching framework presented in Chapter 4, similarly to edge caching, can be easily modelled since each request traverses only a single cache. As a result, this approach has predictable performance and is also robust to variations in traffic patterns. In addition, as we show in section 4.5, the optimal cache placement problem can be solved optimally and in polynomial time for arbitrary topologies.

2.3 Algorithms

After providing a general overview of the caching problem and associated design space, we now discuss more in detail two important types of algorithms used in networked caching systems. These are (i) cache replacement algorithms, which govern the caching decisions made by single caching nodes and (ii) caching meta algorithms, which can be used by caching nodes to make cooperative cache replacement decisions.

2.3.1 Cache replacement algorithms

The design of cache replacement algorithms for content distribution purposes is deeply rooted in previous work on cache replacement algorithms for other computer system applications, such as databases and storage systems. Although such algorithms have been designed for different purposes, their design is suited to content distribution as well, albeit further policies have also been proposed specifically for content distribution. In this section, we provide a thorough overview of cache replacement policies with particular focus on their applicability to content distribution. Unless otherwise specified, we assume that all content items have identical size and denote C as the size of a cache (expressed in number of items) and N as the size of the content catalogue.

It is well known that, independently of demand characteristics, the theoretically optimal cache replacement policy is Bélády's *MIN* algorithm [20]. This algorithm always discards the item that will be requested the farthest away in the future. Although this policy is provably optimal, it requires future knowledge of demand and therefore it is not realisable in practice. It is however useful as a theoretical upper bound of cache performance.

An optimal cache replacement policy is implementable in practice if the demand conforms to the Independent Reference Model (IRM), *i.e.,* the probability of each item being requested is stationary over time and independent of previous requests. In this case, the optimal replacement policy is the *Least Frequently Used (LFU)*, which statically places in cache the C most frequently requested items.

LFU implementation requires content popularity ranking to be known *a priori*. However, even without this information, it can still be implemented by keeping counters for each content requested to learn their request frequency. In this respect, there exist two variants of LFU. The first is *Perfect-LFU*, which maintains counters for every item in the catalogue, even when the item is evicted from the cache. The second is the *In-Cache-LFU*, which drops the counter associated to an item when evicted from the cache. Differently from Perfect-LFU, In-Cache-LFU yields suboptimal performance but has a simpler implementation as a result of the smaller number of counters to be maintained [25].

Despite its optimality, Perfect-LFU is rarely used in practice for two reasons. First, it cannot be implemented in a way that achieves constant time complexity for both lookup and replacement operations. In fact, its most efficient implementation is with a heap, where adding, removing or updating the place of an item has a logarithmic time complexity [25]. Second, since its replacement decisions are based exclusively on frequency without any consideration to recency, it reacts very slowly to variations in content popularity, which frequently occur in practice. Therefore, in operational scenarios LFU is normally used in conjunction with other policies.

To overcome the time complexity and slow convergence time of LFU, a number of replacement policies have been proposed which make replacement decisions based on recency and have $O(1)$ time complexity on both lookup and replacement operations.

The most widely used policy among this family is *Least Recently Used (LRU)*, which replaces the least recently requested item. This policy is normally implemented using a doubly-linked list and operates as follows. When an item currently stored in cache is requested, it is moved to the top of the list. Similarly, upon a request for an item not currently in cache, the requested itme is inserted at the top of the list and the item at the bottom is evicted. LRU has two key advantages that makes it very popular. First, it is very responsive to non-stationary trends, since its replacement decisions are exclusively based on recency. Second, it cannot perform significantly worse than LFU because the ratio between optimal cache hit ratio and LRU cache hit ratio is bounded [65].

Another well known policy albeit less used in practice than LRU is *CLIMB*. In CLIMB, differently from LRU, items are inserted at the bottom of a stack. Each time an item is requested, it is moved one position up in the stack, if not already at the top. Upon a request for an item not stored in cache, the item at the bottom of the stack is replaced. CLIMB is known to yield better performance than LRU under

IRM demand although never formally proved. Furthermore, for a long time a conjecture was held that CLIMB is the optimal finite-memory replacement policy under IRM [14], but it was recently disproved by Gast and Van Houdt [70]. Anyway, the greater steady-state cache hit ratio comes at a cost of slower convergence time and lower reactivity to changes than LRU [78].

Despite its simplicity and ease of implementation, LRU has the drawback of not being well suited for concurrent access. In fact, each lookup resulting in a hit and each replacement both require the insertion of an item at the head of the doubly-linked list. Serialising access to the head of the list may cause contention and detriment performance, especially in highly parallel environments.

Two policies more suitable to concurrent implementations than LRU, although yielding lower cache hit ratio, are *First In First Out (FIFO)* and *Random Replacement (RR)*. According to the FIFO policy, when a new item is inserted, the evicted item is the one which was inserted first in the cache. The behaviour of this policy differs from LRU only when an item already present in the cache is requested. In fact, while in LRU this item would be pushed to the top of the list, in FIFO no movement is performed. Random Replacement (RR) simply replaces a randomly selected item.

Both FIFO and RR policies are more suitable to concurrent implementation than LRU because concurrent access is required only when inserting an item, not at cache hit. They can also be implemented in a simpler way, using a circular queue as opposed to a doubly-linked list. However, they yield worse performance than LRU. In the specific case of IRM demand, it is proven that FIFO and RR yield identical steady-state cache hit ratio [71], although that it is not necessarily true for other types of demand.

Other solutions have been proposed to address the problem of LRU concurrent implementation. One of these is *CLOCK* [45], which approximates LRU behaviour without needing to move an item at each cache hit. Specifically, CLOCK organises items in a circular queue (hence the name CLOCK). Each item is associated to a flag which is originally unset when inserted and is set upon a cache hit. To select an item to replace, CLOCK maintains a pointer used to iterate the circular queue. When iterating the queue, if CLOCK encounters an item whose flag is set, it unsets the flag and moves to the next item, until an item with unset flag is encountered and replaced. At the next replacement operation, the search for the item to replace starts from the position where the last item was replaced. CLOCK is in practice similar to a FIFO cache but with the difference that if an item receives a hit before reaching the bottom of the list, it is not evicted and gets a "second chance".

In addition to concurrency, LRU has the problem of not being scan-resistant, as any scan operation over a large set of unpopular items would thrash the content of the cache. This is a primary concern in databases and disk-based I/O where many legitimate workloads may perform scanning operations and large sequential reads. However, this can be a concern also in networked caching systems as adversarial workloads might perform scanning operations precisely to thrash caches. In addition, content distribution is known to be affected by the problem of *one-timers*, *i.e.,* a large number of items requested only once. For example, Akamai reported that 74% of content objects served by its infrastructure are requested only once [122]. Also EdgeCast observed a similar pattern although in their case one-timers only account for 60% of content items [152]. One-timers have the same effect of scans as both result in a large number of

unpopular items being requested only once, and therefore scan-resistant algorithms can be applied equally well to content distribution.

The poor scan-resistance of LRU can be addressed with variations of the LRU design that incorporate frequency considerations in addition to just recency into replacement decisions.

One simple variation improving LRU resistance to scan is the addition of probabilistic insertion. The resulting algorithm is *q-LRU*, which differs from a pure LRU for the fact that when an item not currently in cache is requested, it is inserted only with probability q. The insertion probability q can be used to tune caching performance. Reducing the value of q increases scan resistance, improves IRM steady-state cache hit ratio but also reduces reactivity to non-stationary trends. It has been proved that q-LRU tends asymptotically to a Perfect-LFU policy when $q \to 0$ [126].

Another approach consists in splitting caching space into separate segments to protect popular contents from scans. This is the approach taken by the *Segmented-LRU (SLRU)* policy [99]. With SLRU, a cache is split into two segments: a probationary and a protected segment, both internally operating according to the LRU policy. When an item is inserted in the cache, it is first inserted at the top of the probationary segment and moved to the protected segment only as a result of a hit. An item reaching the bottom of the protected segment is demoted to the probationary segment, and an item reaching the bottom of the probationary segment is evicted from the cache.

Another policy sharing the same objectives but using a different approach is *LRU/k* [135]. This algorithm replaces the item whose k-th most recent access occurred least recently. As an example, if the k-th most recent access for item A is more recent than the k-th most recent access for item B, then A will be replaced later than B. The evaluations carried out in [135] showed that most of the benefits are achieved for $k = 2$. However, one drawback of LRU/k is its time complexity, which is $O(\log(C))$ for each access.

2Q [97] aims to achieve similar benefits to LRU/2 but with constant time complexity. Its approach is somewhat similar to SLRU. It keeps three buffers, named $\mathrm{A1_{in}}$, $\mathrm{A1_{out}}$ and $\mathrm{A_m}$. $\mathrm{A1_{in}}$ and $\mathrm{A1_{out}}$ are managed according to the FIFO policy while $\mathrm{A_m}$ is managed according to the LRU policy. While both $\mathrm{A1_{in}}$ and $\mathrm{A_m}$ store actual items, $\mathrm{A1_{out}}$ only stores item identifiers. When an item not present in any of the buffers is requested, it is first inserted in $\mathrm{A1_{in}}$. If the content is not requested again, upon eviction from $\mathrm{A1_{in}}$, its identifier is stored in $\mathrm{A1_{out}}$. Upon eviction from $\mathrm{A1_{out}}$ it is discarded. If the content is hit at any time while in $\mathrm{A1_{in}}$ or $\mathrm{A1_{out}}$, it is inserted in $\mathrm{A_m}$, which covers a role similar to the protected segment of the SLRU algorithm.

Another policy similar in principle to 2Q and SLRU is k-LRU which was recently proposed by Martina *et al.* [126]. In practice, with k-LRU a cache consists of a chain of k LRU caches. The first $k-1$ caches store only item identifiers, which require minimal space while the k-th cache stores items. The identifier of an item requested for the first time is stored in the first queue and moved one level above after each hit. Upon eviction from any of these caches, the item or its identifier is discarded from the cache. In practice, with this replacement policy an item needs to be requested at least k times in a reasonably short amount of time to be inserted in the cache.

Akamai use in their operational caches a filtering algorithm similar in principle to 2-LRU, named *cache-after-two-hits* rule [122]. According to this rule, items are inserted in the cache only at the second request. The objective of this algorithm is to prevent insertion of one-timers.

All policies presented above adding scan-resistance to LRU present the problem of requiring manual tuning of a number of parameters that may affect performance. The *Adaptive Replacement Cache (ARC)* algorithm [128] addresses this limitation by adaptively balancing frequency and recency weights affecting replacement decisions. However, despite the good performance and self-tuning properties of ARC, it has seen limited applicability in practice because it is patent encumbered [129].

Finally *CLOCK with Adaptive Replacement (CAR)* [17] revises the design of ARC but builds the algorithm based the CLOCK replacement policy instead of LRU. The result is a more suitable implementation for concurrent access.

2.3.2 Caching meta algorithms

The previous section discussed algorithms that caching nodes can use to make independent replacement decisions. In the context of a network of caches, however, it is beneficial for caching nodes to make replacement decisions cooperatively to maximise network-wide performance metrics, as opposed to each cache selfishly making decisions to maximise their own performance. To achieve this objective, a number of algorithms have been developed allowing nodes to cooperatively make these decisions. These algorithms are normally referred to as *meta algorithms* or *meta policies* [114] to differentiate them from the cache replacement algorithms discussed above. In the remainder of this section, we discuss the main meta algorithms proposed so far.

The simplest and most well known meta algorithm is *Leave Copy Everywhere (LCE)* [114]. According to it, a copy of each content delivered to the requester is replicated at each traversed cache on the path. LCE is generally a good choice in case of flash-crowd events or in case of highly skewed content popularity distributions and it does not require any coordination, either explicit or implicit among caching nodes. However, this policy causes a high degree of redundancy, as all caches along the path are consuming cache resources to hold identical items. To limit this great amount of redundancy and, consequently, to increase the number of distinct content items cached along a delivery path, a number of techniques have been proposed to selectively cache each delivered content in a subset of the nodes of the path.

One of these solutions is *Leave Copy Down (LCD)* [113]. This technique was originally proposed for hierarchical Web caching systems and operates as follows. Whenever a content is served, from either a cache or the origin, the content is replicated only one level down the cache hierarchy on the path towards the requester. This leads to two desirable effects. First, the content is stored only once on the path from serving node to requester, hence mitigating the problem of redundant caching. Second, the content is gradually replicated towards the edge of the network, such that more popular contents are replicated in more nodes and closer to the requesters. This technique requires minimal coordination among caching nodes as they can signal to other nodes downstream whether to cache the content or not by simply appending a flag to the delivered content.

A recent proposal sharing the same objectives of LCD is *"Cache less for more"* [33]. According to this strategy, content objects are cached only once along the path, specifically in the node with the greatest betweenness centrality (*i.e.,* the node with the greatest number of shortest paths traversing it). If more than one node has the maximum value of betweenness centrality, then the content is stored in the node closer to the requester. If this strategy is deployed in very dynamic networks (*e.g.,* ad-hoc mobile networks) where it is challenging for a node to learn its betweenness centrality value, then caching decisions can be made based on the betweenness centrality of its ego-network, *i.e.,* the subnetwork composed of all nodes directly connected to the caching node with an edge. In order for all nodes of the path to determine which is the node with the greatest betweenness centrality, each node writes in the request packet a pair with its identifier and its centrality value, or overwrites any existing one if its centrality value is greater than the one already present in the packet, which was written by a node downstream. This information is then copied in the response packet by the serving node so that all nodes on the delivery path know whether they need to cache the content or not.

Another similar technique is *ProbCache* [139]. This technique also aims at increasing the set of content objects stored over a delivery path. In this proposal, however, the decision on whether to store a content or not is taken randomly instead of deterministically. In fact, each node decides to store a specific content with a certain probability that depends on several factors, including the amount of caching space available upstream and the position of the node in the delivery path. The value of the caching probability computed at each node is engineered in order to maximise content multiplexing over a path. Similarly to LCD and "Cache less for more", ProbCache does not require to keep state at any node and inter-node coordination is achieved by piggybacking a small amount of signalling information on request and response messages.

All three techniques proposed above diversify the content items stored over a path without requiring caching nodes to keep any per-content state and with minimal communication overhead.

2.4 Modelling

After reviewing the design space and algorithms for managing a network of caches, we now focus on the work studying analytical models that describe the operations of such algorithms. Such modelling work has received considerable attention, mainly due to the fact that modelling cache performance, even of simple replacement policies in isolation, such as LRU or FIFO, is very hard. The complexity is further increased when such caches are part of a network.

In this section we start by reviewing the common assumptions adopted with respect to traffic characteristics. We then move to review work analysing the performance of single caches in isolation and, finally, networks of caches.

2.4.1 Common assumptions

Assumptions about demand are normally made with respect to three characteristics: (i) the correlation (or lack thereof) among subsequent requests, (ii) distribution of content popularity and (iii) distribution of content size.

2.4.1.1 Request inter-correlation

The vast majority of caching work, whether in the context of CPUs, databases, disk I/O or content distribution, assumes that the demand conforms to the *Independent Reference Model (IRM)*, originally proposed in [44]. This model assumes that requests are issued over a finite-size catalogue of N items. The probability that each item i, $i = 1, 2, \ldots, N$ is requested, denoted as p_i, is stationary and independent of previous requests.

Despite the fact that operational workloads typically exhibit temporal locality properties [69], the IRM assumption is normally accurate as long as either of the two conditions apply.

1. The cache serves requests received from a large number of requesters which cumulatively mitigate the temporal locality inherent to demand of a single requester [159].

2. The cache churn time is small compared to the timescale of content popularity changes [69], [90].

Recent work, such as [69] and [168], investigated novel traffic models accounting for temporal locality characteristics. Their focus is for very dynamic traffic, where changes occur in the same timescale of cache churn. However, for most cases of practical interest, the IRM assumption is reasonably accurate and also leads to more tractable cache performance models. For this reason, it has been widely used in the literature.

2.4.1.2 Popularity distribution

When assumptions are made regarding the distribution of content popularity, it is normally assumed that it follows a heavy-tailed distribution, since this behaviour was consistently evidenced by past measurement studies on a number of heterogeneous content request workloads [12], [25], [32]. In particular, and especially for content distribution traffic, the standard assumption is that traffic follows a Zipf distribution (sometimes referred to, in some literature, as Zipf-like distribution or Generalised Power Law) [25]. According to this distribution, the probability that an item i is requested is:

$$p_i = \frac{i^{-\alpha}}{H_N^{(\alpha)}} = \frac{i^{-\alpha}}{\sum_{k=1}^{N} k^{-\alpha}} \tag{2.1}$$

where $H_N^{(\alpha)}$ is the generalised N^{th} order harmonic number and $\alpha \geq 0$ is a constant parameter affecting the skewness of the distribution. The greater the value of α, the greater is the skewness of the distribution. In the limit case of $\alpha = 0$, a Zipf distribution becomes a uniform distribution.

The values of α best fitting the traffic in real operational systems vary, but measurement studies report values between 0.6 and 1.2 [25], [58], [85], [160].

It should also be noted that a Zipf distribution also accepts infinite support (*i.e.,* $N \to +\infty$), but only as long as $\alpha > 1$. More generally, a Zipf distribution has finite k-th moments if $\alpha > k$.

For the sake of completeness, it is worth pointing out that some work argued that in certain conditions content popularity may be better modelled by other distributions than Zipf. For example, in [85] Imbrenda *et al.* show that the HTTP traffic measured at a backhaul link of Orange in France is best modelled with a trimodal distribution, with a discrete Weibull for the head, a Zipf for the waist and again a discrete

Weibull (but with different parameters) for the tail. Nevertheless, assuming a Zipf distribution all over the catalogue normally models content popularity fairly accurately.

2.4.1.3 Size distribution

Several studies investigated the size distribution of contents for various workloads [12], [25], [52]. Generally, content size distribution accurately fits a heavy-tailed distribution such as a Pareto or a log-normal distribution. However, none of previous work identified any significant correlation between content popularity and content size [25]. Therefore, for the purpose of cache modelling, the most commonly adopted assumption is that all contents are equally sized and caches are sized in number of items.

2.4.2 Single cache

We survey models for cache replacement policies focusing on the case of a single cache analysed in isolation. Unless otherwise stated, we assume that items are unit-sizes and denote C as the size of the cache and N as the size of the content catalogue and reasonably assume that $C < N$.

All most popular cache replacement policies discussed in Sec. 2.3.1 have the peculiarity that, in spite of their simple implementation, they are very difficult to model, even under largely simplifying assumptions. The only exception is represented by Perfect-LFU, which, as discussed above, is the optimal cache replacement policy under IRM demand. Since Perfect-LFU statically stores the C most popular items, assuming that $p_1 \leq p_2 \leq \ldots \leq p_N$, we can easily compute its cache hit ratio h as:

$$h = \sum_{i=1}^{C} p_i \tag{2.2}$$

For other cache replacement policies, a considerable effort has been made to derive accurate and tractable models, with earliest results dating back to the 70s. To the best of our knowledge, the first work on the subject is by King [104], who derived an exact formulation of the steady-state cache hit ratio of an LRU cache subject to IRM demand. His approach models an LRU cache as a Markov chain with $C! \binom{N}{C}$ states. Despite this model provides exact cache hit ratio results, the extremely large number of states makes it impractical for caches of realistic sizes.

A very important early result is by Gelenbe [71], who proved that under IRM demand FIFO and RR policies have identical steady-state cache hit ratio. A different proof for the same result was recently offered by Martina *et al.* [126]. This equivalence however does not necessarily hold under a demand not conforming to IRM.

Given the complexity of King's LRU model, a number of approximate models have been proposed with lower computational complexity. For example, Dan and Towsley [46] proposed an $O(NC)$ iterative method to approximate the steady-state hit ratio of the LRU replacement policy under IRM demand. This method can be also applied to FIFO replacement policies, but due to the iterative nature, its complexity cannot be estimated. Work from Flajolet *et al.* [59] derived integral expressions for the cache hit ratio of an LRU cache, which can be approximated using numerical integration with a complexity of $O(NC)$.

Che *et al.* [35] made a substantial contribution by proposing a tractable and remarkably accurate model for an LRU cache subject to IRM demand. This model (which in the literature is frequently referred to as *Che's approximation*) introduces the concept of *characteristic time* T, which is the time

an item spends in the cache from insertion to eviction. In an LRU cache subject to IRM demand, T is a random variable and is specific to each content. The intuition of Che's approximation is that, if a cache is large enough, the characteristic times associated to each item fluctuate very little around their mean and therefore they can be accurately approximated with their mean value. In addition, it was observed that the values of the characteristic times of each item are similar and therefore can be approximated with a single constant value for all items of the catalogue. Because of the IRM assumption and the mean-field approximation of T, the following equality holds:

$$\sum_{i=1}^{N} p_{in}(i) = \sum_{i=1}^{N} \left(1 - e^{-p_i T}\right) = C \tag{2.3}$$

where $p_{in}(i)$ is the probability that item i is in cache. From this equation it is then possible to derive the characteristic time T numerically and use it to compute the cache hit ratio of each content $p_{hit}(i)$ as follows:

$$p_{hit}(i) = p_{in}(i) = 1 - e^{-p_i T} \tag{2.4}$$

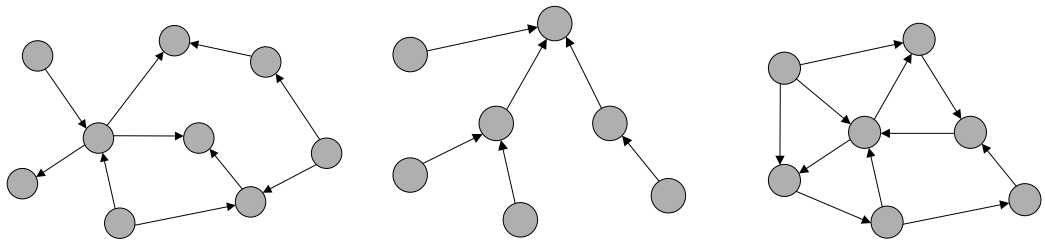
Che's approximation has been extensively adopted in subsequent work (including this thesis) and several extensions have been proposed. Laoutaris [112] extended Che's work by proposing a closed-form approximation of the steady-state cache hit ratio of an LRU cache under Zipf-distributed IRM demand. His approach derives a polynomial approximation of the characteristic time using a Taylor expansion in $T = C$. This method yields accurate results for small caches or popularity distribution with moderate skewness. Fricker *et al.* [66] provided a theoretical justification of the accuracy of Che's approximation beyond its original large cache assumptions and also presented an extension for the RR policy. More recently, Martina *et al.* [126] widely generalised Che's approximation to support a larger set of replacement policies including LFU, FIFO, RR, q-LRU and k-LRU and renewal demand, in addition to IRM.

In another stream of work, Jelenkovic *et al.* made several contributions to the modelling of cache replacement policies, although their work is applicable only limitedly to the case of a Zipf-distributed content popularity with $\alpha > 1$. For example, under these conditions, Jelenkovic [91] provided a closed-form expression for the asymptotic ($C, N \to +\infty$) cache hit ratio of an LRU cache subject to an IRM demand. In the same work, he also showed that the miss probability of an LRU cache is asymptotically at most a factor $e^{\gamma} \approx 1.78$ greater than optimal caching, *i.e.,* Perfect-LFU (with $\gamma = 0.5772\ldots$ being the Euler's constant). The same author made other important contributions regarding the analysis of LRU caches under non-IRM traffic, albeit still under the limitation of Zipf-distributed content popularity with $\alpha > 1$. Specifically, Jelenkovic and Radovanovic showed in [90] and [93] that, under these conditions, an LRU cache subject to a demand with statistical correlation among requests yields a cache hit ratio that is, asymptotically for large cache sizes, the same as in the corresponding LRU with i.i.d. requests. The same authors further investigated in [94] how small can a cache become while still holding the insensitivity property.

2.4.3 Cache networks

The main source of complexity in modelling cache networks stems from the fact that the simplifying assumptions about demand normally used to model a single cache (*e.g.,* IRM demand) do not generally hold when the demand of a cache includes miss streams from other caches, which is the common case in cache networks. In fact, Jelenkovic and Kang [92] showed that the miss stream of an LRU cache subject to an IRM demand presents strong inter-request correlation. However they also showed that the aggregated miss stream from a set of caches each subject to an independent demand tends to IRM if the number of caches is high. As a result, in certain particular cases, such as caching trees with high branching factor, the IRM assumption can still be applied to miss streams, making modelling more tractable.

Before discussing models for cache networks, it is important to distinguish between two types of topologies: *feed-forward* and *arbitrary*. The topology of a cache network is *feed-forward* if on every link the requests flow only in one direction and the requested items flow only in the other direction [144] and the directed graph having as nodes the caches and as directed edges the request flows is a Directed Acyclic Graph (DAG) (see for example Fig. 2.3a and 2.3b). A simple example of a feed-forward topology is a tree with requesters at leafs and a content origin at the root, as shown in Fig. 2.3b. Cache networks which are not feed-forward are defined as arbitrary (see for example Fig. 2.3c).



(a) DAG (feed-forward) (b) Tree (feed-forward) (c) Arbitrary topology

Figure 2.3: Examples of feed-forward and arbitrary cache network topologies

The difference between feed-forward and arbitrary cache network topologies is important for the computation of cache hit ratio because in a feed-forward topology, by definition, the cache hit ratio of a leaf node (*i.e.,* a node subject only to exogenous requests from users, without misses from other caches) does not depend on other caches. As a result, the cache hit ratio of the entire network can be calculated progressively cache-by-cache starting from leaf nodes. Differently, in arbitrary networks, because of the presence of cycles, the miss stream of a cache may influence the demand at the same cache at later time. This makes the analysis of cache performance considerably more complex.

A number of works have investigated the case of feed-forward topologies. For example, Che *et al.* in the paper originally proposing Che's approximation also show how to apply it to the case of a tandem of LRU caches operating under the LCE meta algorithm [35]. Laoutaris *et al.* [113] apply Che's approximation to the case of a tandem of caches operating under the LCD meta algorithm. Finally, Gallo *et al.* [68] study the performance of a network of caches operating under the RR policy. In particular they provide approximations for the miss probability of two specific feed-forward topologies: a line and a

homogeneous tree.

The recent interest in ICN and its envisaged ubiquitous deployment of packet caches on network routers revamped the interest in the modelling of arbitrary networks. Early work by Rosenweig *et al.* [143] proposed an iterative method to approximate the cache hit ratio of an arbitrary network of caches assuming shortest path request routing and LCE content placement. Their method assumes that all requests arriving at each cache are IRM, regardless of whether they are exogenous or not. In spite of this knowingly incorrect assumption, the authors report an error in cache hit ratio estimation in the range 10-15%. Rossini *et al.* [147] later extended this model for the case of ideal Nearest Replica Routing.

A recent stream of work focused on the analysis of TTL caches. In TTL caches, each object is assigned a TTL value, which can be different for different object and also for the same object at different caches of the network. When an object is inserted into a cache as a result of a miss, the TTL is reset and the object is evicted at the expiry of this TTL, irrespective of the hits it received while in cache. Interest in TTL caches is motivated by the fact that results can be applied also to LRU, FIFO and RR policies. TTL caches can in fact accurately mimic their behaviour when configured with appropriate parameters. Early work from Fofack *et al.* [62], [63] derived approximate models for general cache networks of TTL caches. Later work by Berger *et al.* [22] provided two methods to compute exact results for (i) line networks subject to renewal demand and (ii) feed-forward networks subject to Markov Arrival Process (MAP) requests respectively.

More recently Martina *et al.* [126] widely extended Che's approximation to model the performance of an arbitrary network of caches using LCE or LCD meta algorithms.

Finally, it is worth mentioning further work by Rosenweig *et al.* [144] that derives conditions for the ergodicity of cache networks.

2.5 Summary

In this chapter we provided a thorough overview of the state of the art of in-network caching. First, we reviewed the two main applications relying on in-network caching functionalities, namely current Content Delivery Networks (CDN) and envisaged Information-Centric Networking (ICN) architectures. We then overviewed the technical challenges related to the management and operation of complex in-network caching systems and specifically focused on the most commonly used content (re)placement algorithms. We then concluded surveying models describing the behaviour of caches, both in isolation and as part of a network of caches.

This chapter lays the foundations required to understand the contributions of this thesis.

Chapter 3

Theoretical foundations of sharded caching systems

3.1 Introduction

Sharding[1] is a widely used technique to horizontally scale storage and caching systems and to address both processing and storage capacity bottlenecks. According to this technique, a large set of items is partitioned into a set of segments, named *shards*, based on the result of a hash function computed on the identifier of the item. Each shard is then mapped to a physical storage or caching device. This technique practically enables to partition data across members of a cluster and to identify the member of the cluster responsible for a given item by simply computing a hash function. Normally, sharding is used in conjunction with consistent hashing [100] to minimise the remapping of items as a result of cluster members joining or leaving the system.

This technique is widely used in a variety of applications. It is for example ubiquitously implemented in database systems[2], Web caches in enterprise networks [145], [163], CDNs [100], [122] and key-value stores [83]. It is also used to partition large forwarding tables across network cards of a router [138] or across different routers of a network [40]. It has also been applied to single-node key-value store implementations in order to partition items across memory regions and CPU cores [120].

However, despite a considerable amount of previous work investigated the design and implementation of sharded systems, there is little theoretical understanding of sharding properties under realistic conditions. In this chapter, we shed light on the performance of such systems. Our contribution focuses on two main aspects of sharded caching systems: load balancing and caching performance.

With respect to load balancing, we first investigate how skewed distributions of item request frequency and heterogeneous item size affect imbalance. We then focus on understanding how item chunking and frontend caches can be used to mitigate it. Our findings show that the skewness of item popularity distribution considerably increases load imbalance. On the other hand, both item chunking and frontend caches are effective solutions. We derive practical formulas for dimensioning frontend caches for load

[1] The term *sharding* is sometimes referred to, in literature, as *hash partitioning* [184] or *hash routing* [145]. Throughout this thesis, we use these terms interchangeably.

[2] To the best of our knowledge all major database implementations, both relational and NoSQL, currently support sharding.

imbalance reduction.

With respect to caching performance, we analyse how a system of shards, each performing caching decisions independently, behaves and also examine how a layer of frontend caches impacts overall caching performance. Our main finding is that under realistic operational conditions a system of sharded caches yields approximately the same cache hit ratio of a single cache as large as the cumulative size of all shards of the system. This result has important practical implications because it makes possible to model a sharded caching system as a single cache, hence making its analysis considerably more tractable. We use this finding to analyse the interactions between a sharded caching system and various configurations of frontend caches.

The remainder of the chapter is organised as follows. In Sec. 3.2 we introduce the system model adopted throughout this chapter. In Sec. 3.3, we analyse the load imbalance aspects of sharded systems. In Sec. 3.4 we analyse how sharding impacts cache hit performance. Based on these results, in Sec. 3.5 we investigate the cache hit performance of a sharded system in conjunction with a layer of frontend caches. Finally, we draw our conclusions in Sec. 3.6.

3.2 System model

Table 3.1: Summary of notation

Symbol	Notation
N	Number of items
K	Number of shards
K'	Number of frontend caches
C	Cache size
L	Fraction of requests served by a shard
X_i	Variable valued 1 if item i assigned to shard, 0 otherwise
Λ	IRM demand
p_i	Probability of item i being requested
h_i	Cache hit ratio of item i
α	Zipf exponent of item request frequency distribution
T, T_F, T_B	Characteristic time of a generic cache, frontend, backend

In this section we describe the system model adopted throughout this chapter and explain assumptions and notation, which we report in Tab. 3.1.

Consistently with previous work, we assume that item requests conform to the Independent Reference Model (IRM) [44], which, as discussed in Sec. 2.4.1, implies that the probability of each item being requested is stationary and independent of previous requests. Requests are issued for items $1, \ldots, N$ of a fixed catalogue of size N with probability $\Lambda = \{p_1, p_2, \ldots, p_N\}$.

The system comprises K shards, with $K < N$, each equipped with the same amount of caching space of $C < \lfloor N/K \rfloor$ items. We assume that items are mapped uniformly to shards according to a hash

function $f_H : [1 \dots N] \rightarrow [1 \dots K]$. Therefore, we can model the assignment of an item i to a shard using a Bernoulli random variable X_i such that:

$$f_{X_i}(x) = \begin{cases} \frac{1}{K}, & x = 1 \\ 1 - \frac{1}{K}, & x = 0 \end{cases} \tag{3.1}$$

3.3 Load balancing

In this section, we investigate how well the randomised assignment of items spreads load across shards. This analysis applies to both caching and storage systems.

This problem is an instance of a *heavily loaded single-choice weighted balls-in-bins game*. A single-choice balls-in-bins game consists in placing each of N balls into one of K bins chosen independently and uniformly at random (i.u.r.). In the weighted case, each ball has a different weight, which in our case is the probability of an item being requested. In the heavily loaded case, which is the case of our interest, the number of balls is considerably larger than the number of bins, *i.e.,* $N \gg K$.

The study of balls-in-bins games has received considerable attention in the past. Unfortunately, there is very little work investigating the specific case of weighted balls-in-bins games, and, to the best of our knowledge, there is no previous work satisfactorily addressing the specific problem of our interest. There is instead a large amount of work addressing the unweighted case. In this context, Raab and Steger [141] showed that if $N \geq K \log(K)$, the most loaded bin receives $N/K + \Theta(\sqrt{N \log(K)/K})$ balls with high probability[3].

Limiting our analysis to the assumption of uniform popularity distribution would not allow us to understand the load balancing dynamics of sharded systems under realistic operational conditions. In fact, as we show below, skewness in item popularity distribution strongly impacts load imbalance.

In addition to considering realistic item popularity distributions in our model, we also make novel contributions by shedding light on the impact of heterogeneous item size, item chunking and frontend caching on load imbalance.

In line with previous work [163], we quantify load imbalance by using the coefficient of variation of load across shards $c_v(L)$[4], where L is a random variable corresponding to the fraction of requests processed by a single shard. Some other literature (*e.g.,* [83], [100]) instead quantifies load imbalance as the ratio between the load of the most loaded shard and the average load of a shard. The latter metric is normally adopted in experimental work [83], where it can be easily measured, or in theoretical work assuming uniform item popularity distribution [100], where it can be tightly bounded using standard Chernoff arguments. However, in theoretical work assuming non-uniform item popularity distribution like ours, Chernoff bounds cannot be easily applied. As a consequence it is not possible to derive tight bounds on the load of the most loaded shard. For this reason we prefer adopting the coefficient of variation as metric, in line with previous work of this kind [163].

[3] We say that an event A occurs with high probability (w.h.p.) if $Pr[A] \geq 1 - N^{-\theta}$ for an arbitrarily chosen constant $\theta \geq 1$

[4] The coefficient of variation of a random variable is the ratio between its standard deviation and its mean value: $c_v(L) = \sqrt{\mathrm{Var}(L)}/\mathrm{E}[L]$

3.3.1 Base analysis

In this section, we analyse how well the randomised assignment of items to shards spreads load, without making any assumption on item popularity distribution.

Adopting the notation introduced above, we can express the fraction of requests each shard receives as:

$$L = \sum_{i=1}^{N} X_i \, p_i \tag{3.2}$$

Since X_i are i.i.d. random variables, the expected value and variance of L can be calculated as:

$$\mathrm{E}[L] = \mathrm{E}\left[\sum_{i=1}^{N} X_i \, p_i\right] = \sum_{i=1}^{N} \mathrm{E}[X_i] \, p_i = \frac{1}{K} \tag{3.3}$$

$$\mathrm{Var}(L) = \mathrm{Var}\left(\sum_{i=1}^{N} X_i \, p_i\right) = \sum_{i=1}^{N} \mathrm{Var}(X_i) \, p_i^2 = \frac{K-1}{K^2} \sum_{i=1}^{N} p_i^2 \tag{3.4}$$

We can now derive $c_v(L)$ as:

$$c_v(L) = \frac{\sqrt{\mathrm{Var}(L)}}{\mathrm{E}[L]} = \sqrt{K-1} \sqrt{\sum_{i=1}^{N} p_i^2} \tag{3.5}$$

We can immediately observe from Eq. 3.5 that the load imbalance increases proportionally to the square root of the number of shards K and to the skewness of item popularity (quantified by $\sum_{i=1}^{N} p_i^2$). Regarding the impact of item popularity skewness on load imbalance, the minimum value of $c_v(L)$ occurs when all items are equally probable, *i.e.,* $p_i = 1/N \quad \forall i \in [1 \dots N]$, while the maximum is when one item is requested with probability 1 and all other items with probability 0. We can then derive the following bounds for $c_v(L)$.

Theorem 3.1. *Let L be the fraction of requests a shard receives in a system with K shards subject to an arbitrary IRM demand over a catalogue of N items. Then:*

$$\sqrt{\frac{K-1}{N}} \leq c_v(L) \leq \sqrt{K-1} \tag{3.6}$$

Proof. The theorem can be proved immediately by jointly applying Hölder's inequality and Minkowski's inequality to bound $\sum_{i=1}^{N} p_i^2$:

$$\frac{1}{N} \left(\sum_{i=1}^{N} p_i\right)^2 \leq \sum_{i=1}^{N} p_i^2 \leq \left(\sum_{i=1}^{N} p_i\right)^2 \tag{3.7}$$

Substituting $\sum_{i=1}^{N} p_i = 1$ (which holds since p_i is the probability of an item $i \in [1 \dots N]$ being requested), we can rewrite the inequality as:

$$\frac{1}{N} \leq \sum_{i=1}^{N} p_i^2 \leq 1 \tag{3.8}$$

Substituting Eq. 3.5 into Eq. 3.8 and rearranging, we obtain Eq. 3.6 □

The above bounds, derived without assuming anything about item popularity, are however of little practical interest since they can span few orders of magnitude in realistic conditions (*i.e.,* $N \geq 10^6$). In the following section, we derive more useful results by making assumptions on the distribution of item popularity.

3.3.2 Impact of item popularity distribution

We extend our analysis by assuming that item popularity follows a Zipf distribution, which, as discussed in Sec. 2.4.1 is known to model very well item popularity distributions in the scenarios of our interest. According to this distribution, the probability of an item being requested is:

$$p_i = \frac{i^{-\alpha}}{\sum_{i=1}^{N} i^{-\alpha}} = \frac{i^{-\alpha}}{H_N^{(\alpha)}} \tag{3.9}$$

where $H_N^{(\alpha)}$ is the generalised N^{th} order harmonic number and $\alpha > 0$ is a parameter affecting the skewness of the distribution.

In the remainder of this section we provide a closed-form approximation for $c_v(L)$ under a Zipf-distributed demand that we will use to derive the results presented in Sec. 3.3.5.

Theorem 3.2. *Let L be the fraction of requests a shard receives in a system with K shards subject to an IRM demand over a catalogue of N items and item request probability distributed following a Zipf distribution with parameter α. Then:*

$$c_v(L) \approx \begin{cases} \sqrt{K-1}\sqrt{\dfrac{(N+1)^{1-2\alpha}-1}{1-2\alpha}} \cdot \dfrac{1-\alpha}{(N+1)^{1-\alpha}-1} & \alpha \neq \left\{\frac{1}{2}, 1\right\} \\ \dfrac{\sqrt{(K-1)\log(N+1)}}{2(\sqrt{N+1}-1)} & \alpha = \frac{1}{2} \\ \sqrt{\dfrac{N(K-1)}{(N+1)\log^2(N+1)}} & \alpha = 1 \end{cases} \tag{3.10}$$

Proof. We start by deriving an approximated closed-form expression of $H_N^{(\alpha)}$. We do so by approximating $H_N^{(\alpha)}$ with its integral expression evaluated over the interval $[1, N+1]$:

$$\sum_{i=1}^{N} \frac{1}{i^{\alpha}} = H_N^{(\alpha)} \approx \int_1^{N+1} x^{-\alpha} dx = \begin{cases} \frac{(N+1)^{1-\alpha}-1}{1-\alpha}, & \alpha \neq 1 \\ \log(N+1), & \alpha = 1 \end{cases} \tag{3.11}$$

We then use Eq. 3.11 to approximate $\sum_{i=1}^{N} p_i^2$ for the three cases $\alpha \notin \left\{\frac{1}{2}, 1\right\}$, $\alpha = \frac{1}{2}$ and $\alpha = 1$:

$$\begin{aligned} \sum_{i=1}^{N} p_i^2 \Big|_{\alpha \notin \left\{\frac{1}{2},1\right\}} &= \sum_{i=1}^{N} \left(\frac{i^{-\alpha}}{\sum_{j=1}^{N} j^{-\alpha}} \right)^2 = \frac{H_N^{(2\alpha)}}{\left(H_N^{(\alpha)}\right)^2} \\ &\approx \frac{(N+1)^{1-2\alpha}-1}{1-2\alpha} \cdot \frac{(1-\alpha)^2}{\left[(N+1)^{1-\alpha}-1\right]^2} \end{aligned} \tag{3.12}$$

$$\sum_{i=1}^{N} p_i^2 \Big|_{\alpha=\frac{1}{2}} = \sum_{i=1}^{N} \left(\frac{i^{-1}}{\sum_{j=1}^{N} j^{-1}} \right)^2 = \frac{H_N^{(1)}}{\left(H_N^{(\frac{1}{2})}\right)^2} \approx \frac{\log(N+1)}{4\left(\sqrt{N+1}-1\right)^2} \tag{3.13}$$

$$\sum_{i=1}^{N} p_i^2 \Big|_{\alpha=1} = \sum_{i=1}^{N} \left(\frac{i^{-1}}{\sum_{j=1}^{N} j^{-1}} \right)^2 = \frac{H_N^{(2)}}{\left(H_N^{(1)}\right)^2} \approx \frac{N}{(N+1)\log^2(N+1)} \tag{3.14}$$

It should be noted that Eq. 3.13 and Eq. 3.14 could be also obtained by deriving the limits of Eq. 3.12 for $\alpha \to \frac{1}{2}$ and $\alpha \to 1$ respectively applying L'Hôpital's rule.

Finally, applying the approximations of Eq. 3.12, Eq. 3.13 and Eq. 3.14 to Eq. 3.5, we obtain Eq. 3.10. □

To better understand the impact of number of shards and item popularity distribution on load imbalance we show in Fig. 3.1 the value of $c_v(L)$ for various values of α and K for $N = 10^6$. The lines labelled as Model refer to the formulation of $c_v(L)$ of Eq. 3.5 while the markers labelled as Simulation refer to the mean value measured out of 25,000 simulations. We did not plot confidence intervals because they were too small to be distinguishable from point markers.

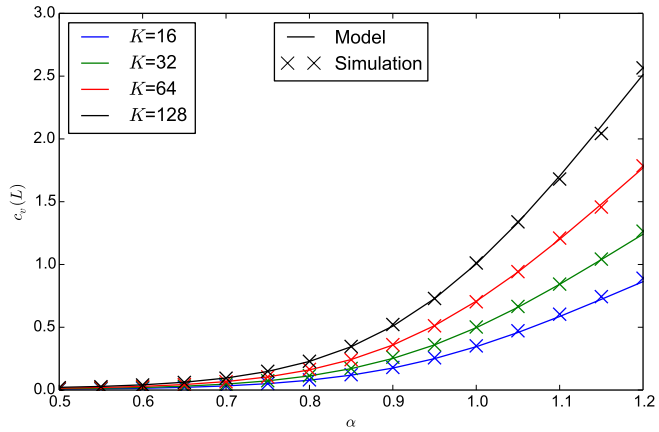


Figure 3.1: Load imbalance $c_v(L)$ *vs.* item popularity skewness (α) and number of shards (K)

Analysing Eq. 3.10 and Fig. 3.1, we can observe that the load imbalance is pretty close to 0 for low values of α, increases as α and K increase and decreases as N increases. In particular, it should be noted that *item popularity skewness considerably affects the load imbalance*. This effect has been already widely experienced in operational systems [79], [83], [131]. Our findings demonstrate that unweighted balls-in-bins analyses investigating purely the number of items assigned to each shard cannot accurately capture load imbalance dynamics under realistic demands.

3.3.3 Impact of heterogeneous item size

If all items have the same size, the fraction of requests processed by a shard corresponds to the fraction of overall traffic it serves. However, if items have heterogeneous sizes, load imbalance in terms of number of requests and amount of traffic served do not coincide anymore. In this section we evaluate the load imbalance in terms of fraction of traffic served assuming that contents have heterogeneous size. Since, as we discussed in Sec. 2.4.1, previous work did not identify any significant correlation between item size and access frequency in the applications of our interest, we assume in our analysis that item size and access frequency are independent.

Theorem 3.3. *Let $L_{(\mu,\sigma)}$ be the throughput served by a shard of a system of K shards subject to an IRM demand $\Lambda = \{p_1, p_2, \ldots, p_N\}$, where the size of each item is a variable with mean μ and variance σ^2.*

Then the coefficient of variation of $L_{(\mu,\sigma)}$ is:

$$c_v\left(L_{(\mu,\sigma)}\right) = \sqrt{K}\sqrt{\left(1-\frac{1}{K}\right)+\frac{\sigma^2}{\mu^2}}\sqrt{\sum_{i=1}^{N} p_i^2} \tag{3.15}$$

Proof. Since we assume that item size and access frequency are independent, we can define $L_{(\mu,\sigma)}$ as:

$$L_{(\mu,\sigma)} = \sum_{i=1}^{N} X_i S_i p_i \tag{3.16}$$

where S_i is an arbitrary random variable with mean μ and variance σ^2, representing the size of item i.

Since we assumed that X_i and S_i are independent we have:

$$\mathrm{E}[X_i \cdot S_i] = \mathrm{E}[X_i] \cdot \mathrm{E}[S_i] = \frac{\mu}{K} \tag{3.17}$$

$$\begin{aligned}\mathrm{Var}(X_i \cdot S_i) &= \mathrm{E}[X_i^2 \cdot S_i^2] - \mathrm{E}[X_i \cdot S_i]^2 \\ &= \mathrm{E}[X_i]^2 \cdot \mathrm{Var}(S_i) + \mathrm{Var}(X_i) \cdot \mathrm{Var}(S_i) + \mathrm{Var}(X_i) \cdot \mathrm{E}[S_i]^2 \\ &= \frac{1}{K}\left[\left(1-\frac{1}{K}\right)\mu^2 + \sigma^2\right]\end{aligned} \tag{3.18}$$

Since $X_1S_1, \ldots, X_NS_N$ are i.i.d., expected value and variance of $L_{(\mu,\sigma)}$ can be calculated as:

$$\mathrm{E}\left[L_{(\mu,\sigma)}\right] = \mathrm{E}\left[\sum_{i=1}^{N} X_i S_i p_i\right] = \sum_{i=1}^{N} \mathrm{E}[X_i S_i]\, p_i = \frac{\mu}{K} \tag{3.19}$$

$$\begin{aligned}\mathrm{Var}\left(L_{(\mu,\sigma)}\right) &= \mathrm{Var}\left(\sum_{i=1}^{N} X_i S_i p_i\right) = \sum_{i=1}^{N} \mathrm{Var}(X_i S_i)\, p_i^2 \\ &= \frac{1}{K}\left[\left(1-\frac{1}{K}\right)\mu^2 + \sigma^2\right]\sum_{i=1}^{N} p_i^2\end{aligned} \tag{3.20}$$

Deriving $c_v\left(L_{(\mu,\sigma)}\right) = \sqrt{\mathrm{Var}\left(L_{(\mu,\sigma)}\right)}/\mathrm{E}\left[L_{(\mu,\sigma)}\right]$ we obtain Eq. 3.15. □

We can observe from Eq. 3.15 that load imbalance still increases proportionally to $\sqrt{K}$ as for the homogeneous item size case. In addition and more importantly, *load imbalance also increases with the coefficient of variation of item size* $c_v(S) = \sigma/\mu$. We can highlight this relation by rewriting Eq. 3.15 as:

$$c_v\left(L_{(\mu,\sigma)}\right) = \Theta\left(\sqrt{K}\left(1+\frac{\sigma}{\mu}\right)\right) = \Theta\left(\sqrt{K}\left(1+c_v(S)\right)\right) \tag{3.21}$$

3.3.4 Impact of chunking

The analysis carried out above shows the emergence of load imbalance as α and K increase. One way to mitigate this is to split items into chunks and map each chunk to a shard independently. Splitting large items into independent chunks and handling each chunk independently is a common practice in many systems. In this section we quantify the benefit of chunking on load balancing.

Theorem 3.4. *Let L_M be the load of a shard in a system where each item is split into M chunks and each chunk is mapped to a shard independently. Chunking reduces the load imbalance by $\sqrt{M}$,* i.e.,

$$c_v\left(L_M\right) = \frac{c_v(L)}{\sqrt{M}} \tag{3.22}$$

Proof. Let assume that chunks of the same item are requested with the same probability and let $X_{i,j}$ be a Bernoulli random variable taking value 1 if chunk j of item i is mapped to the shard under analysis and 0 otherwise. The load at each shard can then be modelled as:

$$L_M = \sum_{i=1}^{N} \sum_{j=i}^{M} X_{i,j} \frac{p_i}{M} = \frac{1}{M} \sum_{i=1}^{N} Y_i \, p_i \tag{3.23}$$

where $Y_i = \sum_{j=1}^{M} X_{j,i} \sim B(M, \frac{1}{K})$ is a binomial random variable.

Since Y_i are i.i.d., $c_v(L_M)$ can be easily calculated as:

$$c_v\left(L_M\right) = \frac{\sqrt{\mathrm{Var}(L_M)}}{\mathrm{E}[L_M]} = \frac{\sqrt{\frac{1}{M^2} \sum_{i=1}^{N} \mathrm{Var}(Y_i)\, p_i^2}}{\frac{1}{M} \sum_{i=1}^{N} \mathrm{E}[Y_i]\, p_i} = \frac{\sqrt{(K-1) \sum_{i=1}^{N} p_i^2}}{\sqrt{M}} \tag{3.24}$$

Substituting Eq. 3.5 into Eq. 3.24, we obtain Eq 3.22. □

Theorem 3.22 shows that *chunking is a very practical and effective solution to reduce load imbalance*. We adopt this technique in the caching node design presented in Chapter 5.

3.3.5 Impact of frontend cache

In this section we investigated whether placing a frontend cache (or an array of frontend caches) in front of a sharded system can reduce the load imbalance experienced by shards.

Intuitively this seems to be the case, since, as we observed above, skewness in item popularity strongly increases load imbalance but at the same time, it can be addressed effectively by caching.

Fan *et al.* [57] already investigated this aspect concluding that a small frontend cache of $O(K \log K)$ items operating according to a Perfect-LFU replacement policy is in fact sufficient to reduce load imbalance. However, their analysis only addresses the load imbalance caused by an adversarial workload with knowledge of the mapping between items and shards attempting to swamp a specific shard.

Our work, differently from [57], addresses the more common case of a non-adversarial workload with Zipf-distributed item popularity. We conclude that a frontend cache is in fact effective in reducing load imbalance and provide closed-form equations for its dimensioning.

For simplicity, the following analysis assumes a single frontend cache, as depicted in Fig. 3.6a. However, it can be trivially shown that the results derived here equally apply to the case of an array of equally sized frontend caches, depicted in Fig. 3.6b, as long as the distribution of item request frequency is identical at all frontend nodes.

Let h_i be the hit probability of item i at a frontend cache. The fraction of requests handled by a shard and its coefficient of variation can be immediately derived as:

$$L = \frac{\sum_{i=1}^{N} X_i \, p_i \, (1 - h_i)}{\sum_{i=1}^{N} p_i \, (1 - h_i)} \tag{3.25}$$

$$c_v(L) = \frac{\sqrt{\mathrm{Var}(L)}}{\mathrm{E}[L]} = \sqrt{K-1}\frac{\sqrt{\sum_{i=1}^{N} p_i^2(1-h_i)^2}}{\sum_{i=1}^{N} p_i(1-h_i)} \tag{3.26}$$

We proceed in our analysis assuming a perfect frontend cache of size $C < N$ that always caches the C most popular items. This corresponds to a Perfect-LFU replacement policy since we assume an IRM workload. We reserve the extension of this analysis to further replacement policies (*e.g.,* LRU) for future work.

Assuming $p_1 \leq p_2 \leq \ldots \leq p_N$, then a perfect frontend cache permanently stores items $1, \ldots, C$. Therefore $c_v(L)$ can be written as:

$$c_v(L) = \sqrt{K-1}\frac{\sqrt{\sum_{i=C+1}^{N} p_i^2}}{\sum_{i=C+1}^{N} p_i} \tag{3.27}$$

We further assume that item popularity follows a Zipf-like distribution with parameter $\alpha > 0$ and apply the approximation of Eq. 3.11 to derive a closed-form equation for $c_v(L)$.

$$c_v(L) \approx \begin{cases} \sqrt{K-1}\frac{\sqrt{\frac{(N+1)^{1-2\alpha}-(C+1)^{1-2\alpha}}{1-2\alpha}}}{\frac{(N+1)^{1-\alpha}-(C+1)^{1-\alpha}}{1-\alpha}} & \text{if } \alpha \notin \{\frac{1}{2}, 1\} \\ \sqrt{K-1}\frac{\sqrt{\log(N+1)-\log(C+1)}}{2(\sqrt{N+1}-\sqrt{C+1})} & \text{if } \alpha = \frac{1}{2} \\ \sqrt{K-1}\frac{\sqrt{\frac{N-C}{(N+1)(C+1)}}}{\log(N+1)-\log(C+1)} & \text{if } \alpha = 1 \end{cases} \tag{3.28}$$

After this preliminary discussion, we now present our main theorem and three corollaries and discuss their results.

Theorem 3.5. *For $\alpha \notin \{\frac{1}{2}, 1\}$, $c_v(L)$ is convex with respect to C. The global minimum of $c_v(L)$ occurs for $C = C^*$ where:*

$$C^* = \underset{C}{\mathrm{argmin}}\, c_v(L) = \gamma(N+1) - 1 \tag{3.29}$$

and γ is the unique solution of:

$$2(1-\alpha)\gamma^{2\alpha-1} - (1-2\alpha)\gamma^{\alpha-1} - 1 = 0 \tag{3.30}$$

in the open interval $\gamma \in (0, 1)$.

Proof. We start by deriving the partial derivative of $c_v(L)$ with respect to C for the case $\alpha \notin \{\frac{1}{2}, 1\}$:

$$\frac{\partial c_v(L)}{\partial C} \approx \frac{\sqrt{K-1}(1-\alpha)(C+1)^{-\alpha}}{2\sqrt{\frac{(N+1)^{1-2\alpha}-(C+1)^{1-2\alpha}}{1-2\alpha}}\left[(N+1)^{1-\alpha}-(C+1)^{1-\alpha}\right]^2} \times \left[2\frac{1-\alpha}{1-2\alpha}\left[(N+1)^{1-2\alpha}-(C+1)^{1-2\alpha}\right] - (C+1)^{-\alpha}\left[(N+1)^{1-\alpha}-(C+1)^{1-\alpha}\right]\right] \tag{3.31}$$

Solving the inequality $\frac{\partial c_v(L)}{\partial C} \geq 0$ and substituting $\gamma = \frac{C+1}{N+1}$ yields:

$$\frac{\partial c_v(L)}{\partial C} \geq 0 \Leftrightarrow \frac{1-\alpha}{1-2\alpha}\left[2(1-\alpha)\gamma^{2\alpha-1} - (1-2\alpha)\gamma^{\alpha-1} - 1\right] \geq 0 \tag{3.32}$$

Now, we analyse the behaviour of $\frac{\partial c_v(L)}{\partial C}$ to demonstrate that $c_v(L)$ has a unique global minimum for $C = C^*$.

Since by definition $0 \leq C < N$, we can immediately observe that $\gamma \in (0, 1)$. More specifically, $\gamma \to 0^+$ when $C \to 0$ and $N \to \infty$, while $\gamma \to 1^-$ when $C \to N$. The value taken by $\frac{\partial c_v(L)}{\partial C}$ at the boundaries of the γ interval are:

$$\lim_{\gamma \to 0^+} \frac{\partial c_v(L)}{\partial C} = \begin{cases} -\frac{1-\alpha}{1-2\alpha} & \text{if } \alpha > 1 \\ -\infty & \text{if } \alpha \in \left(0, \frac{1}{2}\right) \cup \left(\frac{1}{2}, 1\right) \end{cases} \tag{3.33}$$

$$\lim_{\gamma \to 1^-} \frac{\partial c_v(L)}{\partial C} = 0 \quad \forall \alpha \in \mathbb{R}^+ \backslash \left\{\frac{1}{2}, 1\right\} \tag{3.34}$$

We remark that since by definition $\alpha > 0$, then:

$$\lim_{\gamma \to 0^+} \frac{\partial c_v(L)}{\partial C} < 0 \quad \forall \alpha \in \mathbb{R}^+ \backslash \left\{\frac{1}{2}, 1\right\} \tag{3.35}$$

We now investigate under what conditions $\frac{\partial \frac{\partial c_v(L)}{\partial C}}{\partial \gamma} \geq 0$ and observed that:

$$\begin{aligned} \frac{\partial \frac{\partial c_v(L)}{\partial C}}{\partial \gamma} \geq 0 &\Leftrightarrow (1-\alpha)^2 \gamma^{\alpha-2}(-2\gamma^\alpha + 1) \geq 0 \\ &\Leftrightarrow \gamma \leq 2^{-\frac{1}{\alpha}} \end{aligned} \tag{3.36}$$

This shows that $\frac{\partial c_v(L)}{\partial C}$ is negative for $\gamma \to 0^+$, strictly increases for $0 < \gamma < 2^{-\frac{1}{\alpha}}$, reaches a global maximum for $\gamma = 2^{-\frac{1}{\alpha}}$ and then strictly decreases for $2^{-\frac{1}{\alpha}} < \gamma < 1$ tending to 0 for $\gamma \to 1^-$.

From this analysis we can conclude that $\frac{\partial c_v(L)}{\partial C}$ is strictly positive at its global maximum ($\gamma = 2^{-\frac{1}{\alpha}}$). Since $\frac{\partial c_v(L)}{\partial C}$ is continuous over $\gamma \in (0, 1)$, applying the intermediate value theorem we can conclude that there exists at least a value of $\gamma \in (0, 2^{-\frac{1}{\alpha}})$ for which $\frac{\partial c_v(L)}{\partial C} = 0$. Since over that interval $\frac{\partial c_v(L)}{\partial C}$ is strictly increasing, that root is unique and it is a local minimum of $c_v(L)$. Also, this analysis shows that $\frac{\partial c_v(L)}{\partial C}$ cannot have roots for $2^{-\frac{1}{\alpha}} < \gamma < 1$. Therefore, the minimum of $c_v(L)$ is global.

Finally, since we know that the only root of $\frac{\partial c_v(L)}{\partial C}$ is the global minimum of $c_v(L)$, we obtain Eq. 3.30 by simplifying $\frac{\partial c_v(L)}{\partial C} = 0$. □

After presenting our main theorem, we present three corollaries providing simpler closed-form solutions for three specific cases.

Corollary 3.5.1. *For $\alpha > 1$ and $N \to +\infty$, then:*

$$c_v(L) = \sqrt{\frac{K-1}{C+1}} \cdot \frac{\alpha - 1}{\sqrt{2\alpha - 1}} \tag{3.37}$$

Proof. If $\alpha > 1$ and $N \to +\infty$, then $(N+1)^{1-\alpha} \to 0$ and $(N+1)^{1-2\alpha} \to 0$. Applying these substitutions to Eq. 3.28 we obtain Eq. 3.37. □

From Eq. 3.37 it can be observed that $c_v(L)$ is strictly decreasing with respect to C, *i.e.,* increasing C always reduces load imbalance.

Other interesting results, not limited to the asymptotic case $N \to +\infty$, arise from the analysis of the $\alpha = 1$ and $\alpha = \frac{1}{2}$ cases.

Corollary 3.5.2. *If $\alpha = 1$ and $N \geq 4$, $c_v(L)$ is convex with respect to C. The global minimum of $c_v(L)$ occurs for $C = C^*$, where:*

$$C^* = \underset{C}{\operatorname{argmin}}\, c_v(L) = -1 - \frac{1}{2} \cdot W\left(-2e^{-2}\right) \cdot (N+1) \tag{3.38}$$

and W denotes the main branch of the Lambert W function.

If N is large, then:

$$C^* \approx 0.2 \cdot N \tag{3.39}$$

Proof. The proof is analogous to that of Theorem 3.5. We calculate the partial derivative of $c_v(L)$ with respect to C for the case $\alpha = 1$.

$$\frac{\partial c_v(L)}{\partial C} \approx \frac{\sqrt{(K-1)}\left[(N-C) - \frac{1}{2}(N+1)\log\left(\frac{N+1}{C+1}\right)\right]}{(C+1)^2(N+1)\log^2\left(\frac{N+1}{C+1}\right)\sqrt{\frac{N-C}{(N+1)(C+1)}}} \tag{3.40}$$

Solving the inequality $\frac{\partial c_v(L)}{\partial C} \geq 0$ yields:

$$\frac{\partial c_v(L)}{\partial C} \geq 0 \Leftrightarrow C \geq -1 - \frac{1}{2}W\left(-2e^{-2}\right)(N+1) = C^* \tag{3.41}$$

Solving $W(-2e^{-2})$ numerically, we obtain:

$$C^* = 0.2032 \cdot N - 0.797 \tag{3.42}$$

It can be immediately observed that for $N < 3.92$ we have $C^* < 0$ which is not an admissible solution. Therefore, since $N \in \mathbb{N}$ the solution is only valid for $N \geq \lceil 3.92 \rceil = 4$

Finally, for large N the constant part of Eq. 3.42 is negligible and therefore $C^* \approx 0.2032 \cdot N$ □

We conclude by analysing the case for $\alpha = \frac{1}{2}$

Corollary 3.5.3. *If $\alpha = \frac{1}{2}$ and $N \geq 12$, $c_v(L)$ is convex with respect to C. The global minimum of $c_v(L)$ occurs for $C = C^*$, where:*

$$C^* = \underset{C}{\operatorname{argmin}}\, c_v(L) = \frac{N+1}{4\,W_{-1}\left(-\frac{1}{2}e^{-\frac{1}{2}}\right)^2} - 1 \tag{3.43}$$

and W_{-1} denotes the lower branch of the Lambert W function.

If N is large, $C^* \approx 0.08 \cdot N$

Proof. Also in this case the proof is analogous to the one of Theorem 3.5. We calculate the partial derivative of $c_v(L)$ with respect to C for the case $\alpha = \frac{1}{2}$.

$$\frac{\partial c_v(L)}{\partial C} \approx \frac{\sqrt{K-1}\left[\log\left(\frac{N+1}{C+1}\right) - \sqrt{\frac{N+1}{C+1}} + 1\right]}{4\sqrt{C+1}\left(\sqrt{N+1} - \sqrt{C+1}\right)^2\sqrt{\log\left(\frac{N+1}{C+1}\right)}} \tag{3.44}$$

Solving the inequality $\frac{\partial c_v(L)}{\partial C} \geq 0$ yields:

$$\frac{\partial c_v(L)}{\partial C} \geq 0 \Leftrightarrow C \geq \frac{N+1}{4\,W_{-1}\left(-\frac{1}{2}e^{-\frac{1}{2}}\right)^2} - 1 = C^* \tag{3.45}$$

Solving $W_{-1}\left(-\frac{1}{2}e^{-\frac{1}{2}}\right)$ numerically, we obtain:

$$C^* = 0.08 \cdot N - 0.92 \tag{3.46}$$

It can be immediately observed that for $N < 11.34$ we have $C^* < 0$ which is not an admissible solution. Therefore, since $N \in \mathbb{N}$ the solution is only valid for $N \geq \lceil 11.34 \rceil = 12$

Finally, for large N the constant part of Eq. 3.46 is negligible and therefore $C^* \approx 0.0.08 \cdot N$ □

This analysis showed that *a frontend cache is effective in reducing load imbalance in all scenarios of practical interest as long as properly dimensioned*.

From Theorem 3.5 and its corollaries we can observe that the global minimum of $c_v(L)$ does not depend on the absolute values of C or N but on the quantity $\gamma = \frac{C+1}{N+1}$, as well as α. In Fig. 3.2 we plot all values of γ for which we observe a global minimum of $c_v(L)$ for various values of α, which represent the skewness of item popularity distribution. Typical workloads can be modelled with $\alpha \in [0.6, 1.2]$. In that interval, the values of γ that minimise load imbalance are comprised between 0.11 and 0.24. This implies that even in the worst case scenario, the frontend cache should be smaller than 11% of the size of item population to ensure that any addition of caching space reduces load imbalance. In practice such frontend caches are rarely larger than 1% of the item population. Hence, we conclude that *any typical frontend cache reduces load imbalance*.

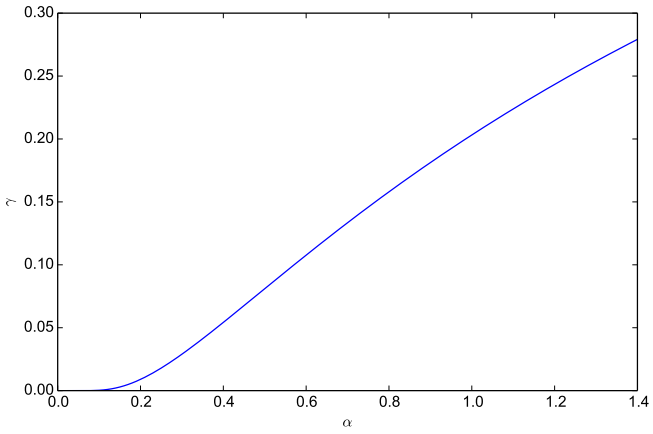


Figure 3.2: Relationship between γ and α

As a further step in our evaluation, we quantify the actual reduction of load imbalance achieved by a frontend cache and plot results in Fig. 3.3 for various values of α and cache/catalogue ratio C/N in the case of $N = 10^6$. In the plot, the lines labelled as *Exact* have been drawn using Eq. 3.27 while the lines labelled as *Model* have been drawn using Eq. 3.28.

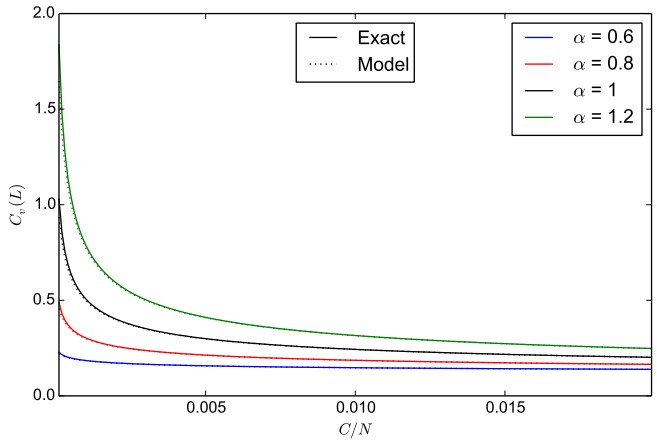


Figure 3.3: Load imbalance $c_v(L)$ in presence of a frontend cache

This analysis allows us to draw three main conclusions. First of all, the load imbalance model, which relies on the integral approximation of the generalised harmonic number of Eq. 3.11, is very accurate. This is demonstrated by the almost perfect agreement between exact and model results. Second, the imbalance reduction is greater for more skewed item distributions. This is expected because, as discussed above, more uniform distributions induce lower load imbalance and therefore there is less improvement that a frontend cache can provide. Finally, and most importantly from a practical standpoint, *even a very small cache of around 0.5% of content catalogue is sufficient to carry out a substantial reduction in load imbalance*. Further increasing cache size still reduces load imbalance but less substantially.

3.4 Cache hit ratio

In this section we focus on investigating the performance of a sharded system in terms of cache hit ratio, assuming that each shard is not a permanent store but evicts items according to a replacement policy independently from other shards. In particular, our objective is to compare the performance of a system of caching shards with that of a single cache with capacity equal to the cumulative capacity of all the shards of the system. We assume that each shard has equal size C, with $C < \lfloor K/N \rfloor$ items and all shards operate according to the same replacement policy.

Our analysis relies on results from *Che's approximation* [35] which we briefly introduced in Sec. 2.4.2 and discuss here more in detail. This approximation introduces the concept of *characteristic time* T of a cache, which corresponds to the time spent by an item in a cache from insertion to eviction. The characteristic time is a random variable specific to each item. However, approximating it with a single constant value for all items still yields very accurate results [66] and widely simplifies performance analysis because it decouples the dynamics of a specific item from all other items.

Definition 3.1. The *characteristic time* T of a cache subject to an IRM demand is the average number of item requests required to evict an item from the cache.

Because of the IRM demand assumption, the probability that an item is in the cache p_{in} is equal by definition to its hit probability p_{hit}, as a consequence of the arrival theorem of a Poisson process.

Applying Che's approximation, it is possible to determine the hit ratio of each item i simply knowing the request probability of that item p_i and the characteristic time of the cache T. In the specific case of the LRU replacement policy this corresponds to:

$$p_{hit}(p_i, T) = p_{in}(p_i, T) = 1 - e^{-p_i T} \tag{3.47}$$

The generalisation of Martina *et al.* [126] shows that this method can be applied to a larger variety of cache replacement policies, including LFU, FIFO, RANDOM and q-LRU. We refer the reader to [126] for equations to derive hit ratios for these replacement policies.

Definition 3.2. A cache replacement policy is *defined by characteristic time* if the steady-state per-item hit ratio of a cache operating under such policy subject to an IRM demand is exclusively a function of the request probability of the item and the characteristic time of the cache.

Finally, the characteristic time of a cache T can be computed by solving the following equation:

$$\sum_{i=1}^{N} p_{in}(p_i, T) = C \tag{3.48}$$

For all cache replacement policies listed above, this equation does not have a closed-form solution, but can be easily solved numerically.

After these initial considerations, we present the following theorem and devote the remainder of this section to prove it and to discuss its applicability in realistic operational conditions.

Theorem 3.6. *Let $\mathcal{C}_{\mathcal{S}}$ be a caching shard of capacity C belonging to a system of K shards, all with the same capacity and operating under the same replacement policy $\mathcal{R}$ defined by characteristic time. The overall system is subject to an IRM demand $\Lambda = \{p_1, p_2, \ldots, p_N\}$ and items $1, \ldots, N$ are mapped to shards $1, \ldots, K$ uniformly. Then, for large C, the cache hit ratio of $\mathcal{C}_{\mathcal{S}}$ is approximately equal to the cache hit ratio of a single cache of size $K \cdot C$ operating under replacement policy $\mathcal{R}$ subject to the demand Λ.*

Proof. We denote T_S as the characteristic time of a shard of size C and T as the characteristic time of a single cache of size $K \cdot C$ subject to the same demand of the overall sharded system. Assuming uniform assignment of items to shards and applying Eq. 3.48 we obtain:

$$\psi\left(T_S\right) = \sum_{i=1}^{N} p_{in}(X_i\, p_i, T_S) = C \tag{3.49}$$

$$\phi(T) = \sum_{i=1}^{N} p_{in}(p_i, T) = K \cdot C \tag{3.50}$$

where $\psi\left(T_S\right)$ is a random variable whose outcome depends on X_i, while $\phi\left(T\right)$ is constant. It should be noted that $p_{in}(\cdot)$ in Eq. 3.49 and 3.50 are the same function because both caches operate according to the same replacement policy and p_{in} depends only on p_i and T because we assumed that such policy is determined by characteristic time.

By definition, if an item is never requested it cannot be in the cache, hence $p_{in}(0,T) = 0$. Since $X_i \in \{0,1\}$, then $p_{in}(X_i\, p_i, T_S) \equiv X_i\, p_{in}(p_i, T_S)$. We can then rewrite Eq. 3.49 as:

$$\psi\left(T_S\right) = \sum_{i=1}^{N} X_i\, p_{in}(p_i, T_S) = C \tag{3.51}$$

Let $\overline{T}_S$ be the value of T_S for which $\mathrm{E}[\psi(T_S)] = C$. Therefore:

$$E\left[\psi\left(\overline{T}_S\right)\right] = \mathrm{E}\left[\sum_{i=1}^{N} X_i p_{in}(p_i, \overline{T}_S)\right] = \frac{1}{K}\sum_{i=1}^{N} p_{in}(p_i, \overline{T}_S) = C \tag{3.52}$$

We now proceed showing that $T_S \approx \overline{T}_S$ for large C. First, we note that since by definition $p_{in}(p_i, \overline{T}_S) \in [0,1]$, $\forall i \in [1 \ldots N]$, the following inequality holds:

$$\sum_{i=1}^{N} \left[p_{in}(p_i, \overline{T}_S)\right]^2 \leq \sum_{i=1}^{N} p_{in}(p_i, \overline{T}_S) \tag{3.53}$$

Jointly applying this inequality and Eq. 3.52, we derive the following upper bound of $\mathrm{Var}(\psi(\overline{T}_S))$:

$$\begin{aligned}
\mathrm{Var}\left(\psi\left(\overline{T}_S\right)\right) &= \mathrm{Var}\left(\sum_{i=1}^{N} X_i\, p_{in}(p_i, \overline{T}_S)\right) \\
&= \frac{K-1}{K^2}\sum_{i=1}^{N} \left(p_{in}(p_i, \overline{T}_S)\right)^2 \\
&\leq \frac{K-1}{K^2}\sum_{i=1}^{N} p_{in}(p_i, \overline{T}_S) \\
&= \left(1 - \frac{1}{K}\right) C
\end{aligned} \tag{3.54}$$

Then, jointly applying Chebyshev's inequality and Eq. 3.54 to $\psi(\overline{T}_S)$ we obtain:

$$\begin{aligned}
P(\left|\psi(\overline{T}_S) - E\left[\psi(\overline{T}_S)\right]\right| \geq \epsilon C) &\leq \frac{\mathrm{Var}(\psi(\overline{T}_S))}{\epsilon^2 C^2} \\
&\leq \frac{1 - 1/K}{\epsilon^2 C^2} E\left[\psi(\overline{T}_S)\right] \\
&= \left(1 - \frac{1}{K}\right)\frac{1}{\epsilon^2 C}
\end{aligned} \tag{3.55}$$

From Eq. 3.55, we can immediately observe that fluctuations of $\psi(\overline{T}_S)$ around its mean increase proportionally to the number of shards K and decrease proportionally to the cache size C. Even assuming the very unfavourable case of $K \to +\infty$ and small cache size, for example $C = 10^6$, the probability that the mean field approximation induces an error above $0.01C$ is 1%. If single caches of the system are large enough and/or there is a reasonably small number of shards, $\psi(\overline{T}_S)$ fluctuates very little around its mean. Therefore, for the operating conditions of our interest, we can reasonably assume that $T_S \approx \overline{T}_S$.

Applying this result to Eq. 3.52 and rearranging, we obtain:

$$\sum_{i=1}^{N} p_{in}(p_i, T_S) = K \cdot C \tag{3.56}$$

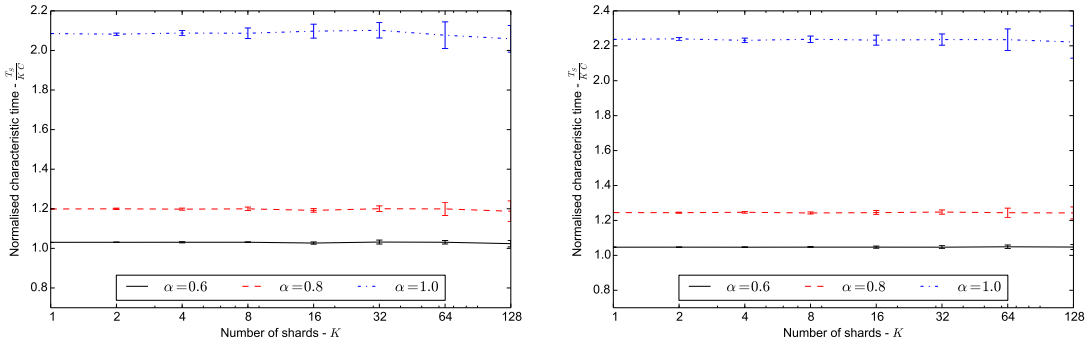
Finally, comparing Eq. 3.56 with Eq. 3.50, it can be immediately proved that the characteristic time of a cache of size C in a system of K caching shards is approximately equal to the characteristic time of a

single cache of $K \cdot C$ size. Therefore:

$$T_S^{(C)} = \psi^{-1}(C) \approx \phi^{-1}(KC) = T^{(KC)} \tag{3.57}$$

□

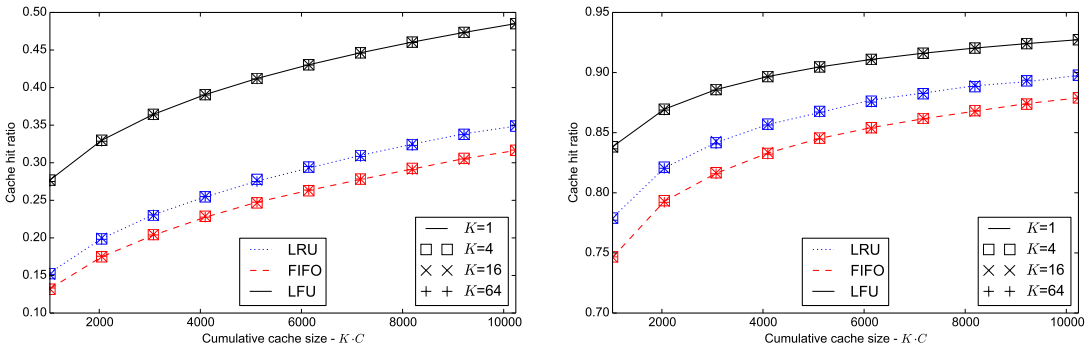
We further validate the approximation of Theorem 3.6 numerically and draw results in Fig. 3.4, where we show the value of T_S, normalised by the cumulative cache size, for different values of K and with item popularity distributed according to Zipf distributions with exponents $\alpha \in \{0.6, 0.8, 1.0\}$. The results are consistent with our analysis. In all cases considered, the mean value of T_S is not affected by variations in number of shards K and is the same in the case of a single cache ($K = 1$). The standard deviation of T_S, represented by the error bars, increases with K, as expected, but it remains low in comparison to the mean value of T_S as it is possible to see from the values of the y-axis.



(a) LRU (b) FIFO/RANDOM

Figure 3.4: Characteristic time of a shard (T_S) *vs.* number of shards (K) and α

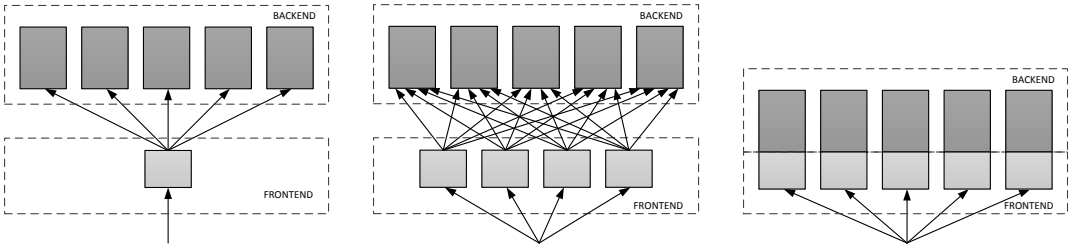
The insensitivity of T_S against K is also expectedly reflected in cache hit ratio performance. We simulate systems of caching shards under the LRU, FIFO and Perfect-LFU replacement policies for various values of cache size, α and K and plot results in Fig. 3.5. The graphs show again insensitivity of cache hit ratio with respect to K, further validating our analysis.



(a) $\alpha = 0.8$ (b) $\alpha = 1.2$

Figure 3.5: Cache hit ratio for a single cache ($K = 1$) and systems of shards ($K \in \{4, 16, 64\}$)

3.5 Two-layer caching



(a) Single frontend cache (b) Array of frontend caches (c) Co-located frontend/backend

Figure 3.6: Two-layer caching deployment models

Sharded caching systems are sometimes deployed preceded by a layer of frontend caches. This layer of caches affects not only the load balancing (as we observed in Sec. 3.3.5) but also the caching performance of the backend.

In this section, we provide models to analyse the performance of three common frontend cache deployments: a single frontend (Fig. 3.6a), an array of frontends (Fig. 3.6b) and co-located frontends and backends (Fig. 3.6c)

The results presented in this section rely on Theorem 3.6 which shows how a cluster of sharded caches behaves like a single cache with size equal to the cumulative size of all shards. We also rely in our analysis on the framework proposed in [126] for modelling the performance of networks of caches operating under replacement policies defined by characteristic time.

Throughout this section we assume that the system is subject to an IRM demand $\Lambda = \{p_1, p_2, \ldots, p_N\}$. For practical reasons, we assume that both frontend and backend caches operate according to an LRU replacement policy, because of its widespread adoption. However this analysis can be easily extended to additional cache evictions policies defined by characteristic times by using different formulas to compute the probability that an item is in a cache. In the cases where there are multiple frontend caches, we assume that each of them has the same size. We denote the size of a single frontend cache as C_F and the cumulative size of the backend (*i.e.,* the sum of the sizes of each shard) as C_B. Similarly, we denote the characteristic time of a single frontend cache and the backend sharded system as T_F and T_B respectively. We also reasonably assume that the cumulative size of the backend cache C_B is larger than each frontend cache C_F, hence $C_B > C_F$ and therefore $T_B > T_F$. This assumption obviously holds in real systems because if each frontend size was larger than the backend, then the backend would never be hit.

3.5.1 Single frontend cache

In this scenario, depicted in Fig. 3.6a, we assume a single frontend cache. This is for example the deployment model used by FAWN-KV [7]. Since, as mentioned above, we can model the sharded system as a single cache, the entire system can be modelled as a tandem of caches (the frontend and the backend caches).

Recalling the definition of characteristic time, we can observe that the probability that an item is

found at the frontend cache corresponds to the probability that the item was requested in the interval $[t - T_F, t]$ independently of the state of the backend cache. Differently, an item is found at the backend cache only if it was requested during the interval $[t - T_B, t - T_F]$ assuming, as we did, that $T_B > T_F$.

We can immediately start deriving the cache hit ratio at the frontend since it does not depend on the state of the backend. We compute its characteristic time and subsequently its cache hit ratio for item i (which we denote as $h_i^{(F)}$) by applying Eq. 3.48 and Eq. 3.47:

$$\sum_{i=1}^{N} 1 - e^{-p_i T_F} = C_F \tag{3.58}$$

$$h_i^{(F)} = 1 - e^{-p_i T_F} \tag{3.59}$$

The average rate of requests missing the frontend cache can then be written as:

$$\overline{p}_i^{(B)} = p_i \left(1 - h_i^{(F)}\right) \tag{3.60}$$

It should be noted however, that in this case, requests are no longer IRM, since they are not exogenous requests but are misses from an LRU cache, which are known to be correlated [92]. For this reason we use the overline notation.

We then derive the characteristic time T_B of the backend cache subject to the miss stream from the frontend. This operation provides approximate results because the miss stream of the frontend is not IRM and therefore Che's approximation does not hold. It has however been shown in [126] that even if the IRM assumption does not hold, this method still yields accurate results and we further validate this with simulations in Sec. 3.5.4.

$$\sum_{i=1}^{N} 1 - e^{-\overline{p}_i^{(B)} T_B} = C_B \tag{3.61}$$

We then derive the cache hit ratio at the backend by applying Eq. 3.47 and recalling that a backend hit for an item occurs only if it was requested during the interval $[t - T_B, t - T_F]$:

$$h_i^B \approx 1 - e^{-\overline{p}_i^{(B)}(T_B - T_F)} \tag{3.62}$$

Finally, we can derive the overall cache hit ratio of the two-layer system as:

$$\begin{aligned} h &= \sum_{i=1}^{N} p_i \left[1 - \left(1 - h_i^{(F)}\right)\left(1 - h_i^{(B)}\right)\right] \\ &= 1 - \sum_{i=1}^{N} p_i \cdot \exp\left[-\left(p_i T_F + p_i^{(B)}(T_B - T_F)\right)\right] \end{aligned} \tag{3.63}$$

3.5.2 Array of frontend caches

This case, depicted in Fig. 3.6b, is similar to the first case except that, instead of a single frontend, it comprises an array of caches, where each request can arrive at any of these caches randomly.

We model the caching system as a two-level tree with the system of shards as root and the frontends as leaves. We assume that all K' frontend caches have the same size C_F. We also assume that

requests are uniformly distributed among frontend caches. As a result, since the global demand is $\Lambda = \{p_1, p_2, \ldots, p_N\}$, the demand at each frontend cache is $\Lambda_{K'} = \{p_1/K', p_2/K', \ldots, p_N/K'\}$.

Similarly to the previous case, a hit at the frontend cache occurs only if an item was requested during the interval $[t - T_F, t]$. However, differently from the previous case, a cache hit at the backend on an item received from a specific frontend F_j occurs only if either a request from F_j arrived in the interval $[t - T_B, t - T_F]$ or if at least one request arrived from any other frontend cache $F_{k,k\neq j}$ in the interval $[t - T_B, t]$.

Similarly to the single frontend case, we can start deriving the characteristic time T_F and the hit ratio $h_i^{(F)}$ at each frontend cache applying Eq. 3.48 and Eq. 3.47.

$$\sum_{i=1}^{N} 1 - e^{-\frac{p_i}{K'}T_F} = C_F \tag{3.64}$$

$$h_i^{(F)} = 1 - e^{-\frac{p_i}{K'}T_F} \tag{3.65}$$

Analysing Eq. 3.64 it can be immediately noted that T_F/K' is constant. As a result (and expectedly), the cache hit ratio of each frontend cache $h_i^{(F)}$ does not depend on the number of frontend caches K'. Finally comparing Eq. 3.64 with Eq. 3.58, it can also be noted that the T_F/K' of Eq. 3.64 is identical to the value T_F of Eq. 3.58, which is also expected since an array of frontend caches with $K' = 1$ corresponds to a single frontend cache. Similarly to the single frontend case, we can proceed deriving the average rate of requests for item i missing the layer of fronted caches, denoted as $\overline{p}_i^{(B)}(j)$ and then the characteristic time of the backend T_B:

$$\overline{p}_i^{(B)} = \sum_{j=1}^{K'} \frac{p_i}{K'} \left(1 - h_i^{(F)}\right) = p_i \left(1 - h_i^{(F)}\right) \tag{3.66}$$

$$\sum_{i=1}^{N} 1 - e^{-\overline{p}_i^{(B)}T_B} = C_B \tag{3.67}$$

Again, it should be noted by comparing Eq. 3.66 and 3.67 with Eq. 3.61 and 3.60 that the values T_B of the single and array cases are identical.

To derive the hit ratio at the backend $h_i^{(B)}$, we apply results from [126] which allow us to compute $h^{(B)}(i|F_j)$, *i.e.,* the hit ratio of item i at the backend, given that it has been forwarded by F_j. Specifically:

$$h^{(B)}\left(i|F_j\right) \approx 1 - e^{-A_{i,j}} \tag{3.68}$$

where:

$$A_{i,j} = \frac{1}{K'} \cdot \overline{p}_i^{(B)}(j) \cdot \max(0, T_B - T_F) + \sum_{k\neq j} \frac{1}{K'} \cdot \overline{p}_i^{(B)}(k) \cdot T_B \tag{3.69}$$

Since all frontend caches are statistically identical and are subject to identical demand, their miss streams are identical. As a result, the rate of requests reaching the backend from each frontend are identical, *i.e.,* $\overline{p}_i^{(B)} = \overline{p}_i^{(B)}(1) = \overline{p}_i^{(B)}(2) = \cdots = \overline{p}_i^{(B)}(K')$. In addition, the hit probability of an

item at the backend does not depend on the specific frontend from which the request is received, *i.e.,* $h^B(i|F_j) = h_i^{(B)}$. We can therefore apply these findings and rewrite Eq. 3.68 as:

$$h_i^{(B)} \approx 1 - e^{-\overline{p}_i^{(B)}\left(T_B - \frac{T_F}{K'}\right)} \tag{3.70}$$

We can finally derive the overall cache hit ratio h by substituting in Eq. 3.63 the values of $h_i^{(F)}$ and $h_i^{(B)}$ o Eq. 3.65 and Eq. 3.70.

$$\begin{aligned} h &= \sum_{i=1}^{N} p_i \left[1 - \left(1 - h_i^{(F)}\right)\left(1 - h_i^{(B)}\right)\right] \\ &= 1 - \sum_{i=1}^{N} p_i \cdot \exp\left[-\left(p_i \frac{T_F}{K'} + p_i^{(B)}(T_B - \frac{T_F}{K'})\right)\right] \end{aligned} \tag{3.71}$$

Analysing 3.70 and recalling that T_F/K' is constant and equal to the T_F of the single frontend case, we can observe that *the cache hit ratio at the backend does not depend on the number of frontend caches*. Since, as shown above, the frontend cache hit ratio is independent of the number of frontends, *the cache hit ratio performance of a system of sharded caches with an array of frontends is identical to that of the single frontend case*.

One difference that is important to highlight though is that, since the miss stream of the frontend caches are independent [92], the greater the value of K' the less is the dependence among requests in the aggregate miss stream. As a result, the Poisson approximation used in Eq. 3.70 becomes more accurate. However, as we show in Sec. 3.5.4, even for low values of K' our model is still very accurate.

3.5.3 Co-located frontend and backend caches

The final configuration, depicted in Fig. 3.6c, is the case in which each backend node reserves a fixed fraction of its caching space to cache content indiscriminately. In this case, item requests are randomly redirected to one of the caches. If the cache is not responsible for the item and it does not have it in the fraction of cache allocated as frontend cache, it forwards the request to the responsible cache and caches the response in the area reserved as frontend cache.

This configuration is similar to the case of an array of frontend with $K' = K$ but with a key difference: frontends never cache items for which its co-located backend is responsible.

The key to understand the modelling of this configuration is the following. Let us assume that requests are uniformly redirected to one of the K frontend caches. With probability $\frac{1}{K}$, a request is forwarded directly to a frontend colocated with the backend node responsible for the item requested. In this case, the request does not trigger a lookup in the frontend but it is processed by the backend directly (which, we remind, is co-located). Therefore, user-generated requests will be forwarded directly to the backend $1/K$ of the times and to each of the K frontends not co-located with the backend responsible for the item with probability $\frac{K-1}{K^2}$.

Similarly to the case of array of frontends, the demand at each frontend (or if co-located with a responsible backend) is $\Lambda_K = \{p_1/K, p_2/K, \ldots, p_N/K\}$. It should also be noted that each frontend cache will only cache a subset of the items catalog approximately of size $N(K-1)/K$, *i.e.,* the items for which the co-located backend is not responsible. Applying the results of Theorem 3.6, we can derive T_F

(assuming an LRU eviction policy) by numerically solving the following equation:

$$\sum_{i=1}^{N} 1 - e^{-\frac{p_i}{K} T_F} = \frac{K}{K-1} C_F \tag{3.72}$$

We can now derive the hit ratio at each frontend cache $h_i^{(F)}$ (which is also identical to the cache hit ratio of the entire frontend layer) and the average rate of requests missing the frontend caches $\overline{p}_i^{(F,B)}$ as in the two previous cases.

$$h_i^{(F)} = 1 - e^{-\frac{p_i}{K} T_F} \tag{3.73}$$

$$\overline{p}_i^{(F,B)} = \sum_{j=1}^{K} \frac{p_i}{K} \left(1 - h_i^{(F)}\right) = p_i \left(1 - h_i^{(F)}\right) \tag{3.74}$$

Recalling that, on average, $1/K$ of the requests reach a backend directly and that $\frac{K-1}{K^2}$ reach each of the K frontends, we can derive the rate of requests for content i reaching a backend directly $p_i^{(D)}$ and after going through a frontend first $p_i^{(F)}$ as:

$$p_i^{(D)} = \frac{p_i}{K} \tag{3.75}$$

$$p_i^{(F)} = \frac{K-1}{K^2} p_i \left(1 - h_i^{(F)}\right) = \frac{K-1}{K^2} \overline{p}_i^{(F,B)} \tag{3.76}$$

Therefore, the average rate of requests reaching the backend (either as a result of a frontend miss or an exogenous backend request) can then be written as:

$$\overline{p}_i^{(B)} = p_i^{(D)} + K p_i^{(F)} = \frac{1}{K} p_i + \frac{K-1}{K} \overline{p}_i^{(F,B)} = p_i \left(1 - \frac{K-1}{K} h_i^{(F)}\right) \tag{3.77}$$

With this information, we derive the characteristic time of the backend T_B by solving numerically the following equation:

$$\sum_{i=1}^{N} 1 - e^{-\overline{p}_i^{(B)} T_B} = C_B \tag{3.78}$$

We can now proceed deriving the cache hit ratio at the backend. Differently from the case of an array of standalone frontend caches however, the hit probability at the backend depends on where the request is received from, *i.e.,* directly to the backend or as a result of a miss from a frontend. Specifically:

$$h_i^{(B)} = \sum_j h_B(i|j) P_j(i) \tag{3.79}$$

where $P_j(i)$ is the probability that a request for item i reaching the backend is forwarded from j, which can represent either a frontend or the exogenous demand (*i.e.,* direct traffic not hitting frontend cache). $P_i(j)$ can be easily derived from Eq. 3.75 and 3.76 and written as:

$$P_i(j) = \begin{cases} \dfrac{p_i^{(D)}}{p_i^{(D)} + K \cdot p_i^{(F)}} = \dfrac{1}{1 + (K-1)\left(1 - h_i^{(F)}\right)} = \dfrac{1}{1+\eta_i}, & \text{if } j = D \\ \dfrac{p_i^{(F)}}{p_i^{(D)} + K \cdot p_i^{(F)}} = \dfrac{(K-1)\left(1 - h_i^{(F)}\right)}{K\left[1 + (K-1)\left(1 - h_i^{(F)}\right)\right]} = \dfrac{\eta_i}{K(1+\eta_i)}, & \text{if } j \in F \end{cases} \tag{3.80}$$

where $\eta_i = (K-1)(1-h_i^{(F)})$.

As in the case of the array of frontend caches, we can derive $h_B(i|j)$ applying results from [126]:

$$h_B(i|j) \approx 1 - e^{-A_{i,j}} \tag{3.81}$$

where

$$A_{i,j} = \lambda_{i,j} \cdot \max(0, T_B - T_F) + \sum_{k \neq j} \lambda_{i,k} \cdot T_B \tag{3.82}$$

and $\lambda_{i,j}$ corresponds to the rate of traffic for item i forwarded from j. This corresponds to $p_{i,F}$ if j is a frontend and $p_{i,D}$ if j is the source of request (traffic reaching the backend directly). Naturally $A_{i,j}$ takes different values depending on whether j is a frontend cache or a source of exogeneous demand. If j is a frontend:

$$\begin{aligned} A_{i,j\in F} &= \frac{K-1}{K^2}\overline{p}_i^{(F,B)}\left[T_B - T_F\right] + \left(\frac{K-1}{K}\right)^2 \overline{p}_i^{(F,B)} T_B + \frac{1}{K} p_i T_B \\ &= \frac{K-1}{K}\overline{p}_i^{(F,B)}\left[T_B - \frac{1}{K}T_F\right] + \frac{1}{K} p_i T_B \\ &= \frac{1}{K} p_i \left[(K-1)\left(1-h_i^{(F)}\right)\left(T_B - \frac{1}{K}T_F\right) + T_B\right] \\ &= \frac{1}{K} p_i \left[\eta_i \left(T_B - \frac{1}{K}T_F\right) + T_B\right] \end{aligned} \tag{3.83}$$

and j is a direct source of traffic:

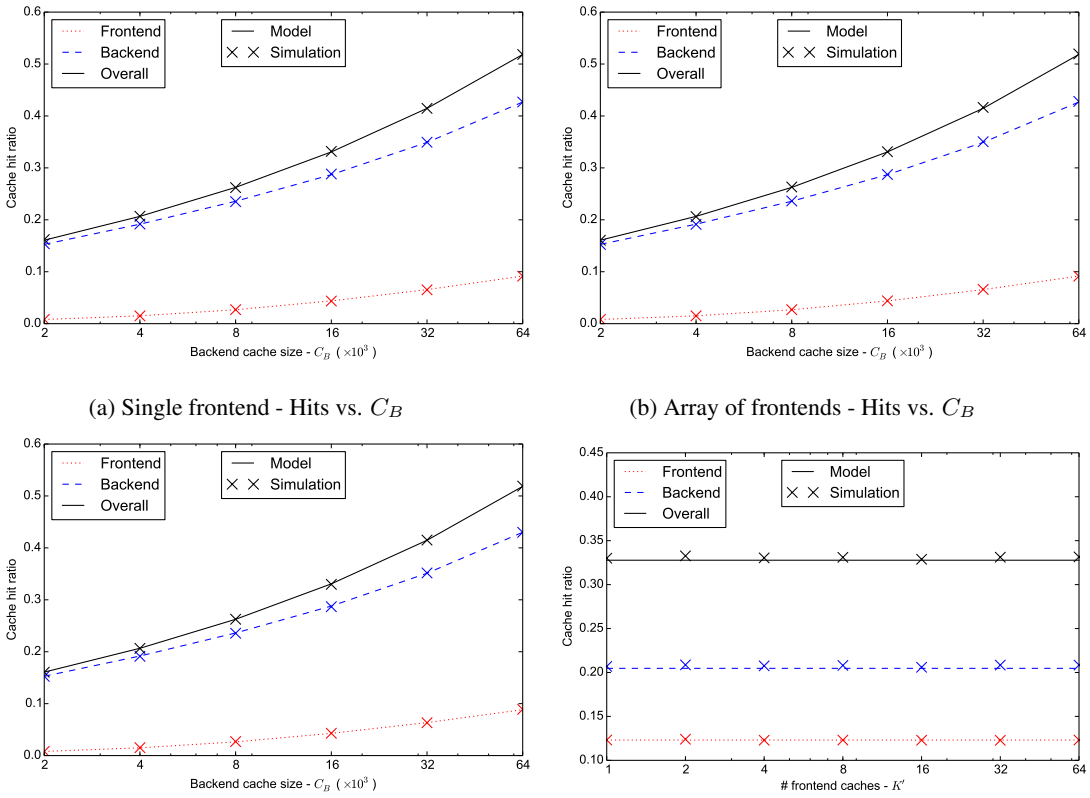
$$\begin{aligned} A_{i,j=D} &= \frac{1}{K}\, p_i\, T_B + \frac{K-1}{K}\, \overline{p}_i^{(F,B)}\, T_B \\ &= \frac{1}{K}\, p_i\, T_B \left[1 + (K-1)\left(1-h_i^{(F)}\right)\right] \\ &= \frac{1}{K}\, p_i\, T_B\, (1+\eta_i) \end{aligned} \tag{3.84}$$

Finally, putting it all together, we can write the cache hit ratio at the backend as:

$$\begin{aligned} h_i^{(B)} &= h_B(i|j=D)\cdot P(j=D) + K \cdot h_B(i|j\in F)\cdot P_i(j\in F) \\ &= \left(1 - e^{-A_{i,j=D}}\right)\cdot P_i(j=D) + K\cdot\left(1-e^{-A_{i,j\in F}}\right)\cdot P_i(j\in F) \\ &= \frac{\eta_i}{1+\eta_i}\left[1 - \exp\left(-\frac{p_i}{K}\left(T_B(1+\eta_i) - \eta_i\frac{T_F}{K}\right)\right)\right] + \frac{1}{1+\eta_i}\left[1-\exp\left(-\frac{p_i}{K}T_B(1+\eta_i)\right)\right] \end{aligned} \tag{3.85}$$

It should be noted that $h_i^{(B)}$ refers to the hit ratio from items stored in the backend caching space and it includes also hits from requests forwarded to the backend directly. In certain cases, it may be preferable to classify direct backend hits as frontend hits because in fact they do not incur the cost of a redirection from the first contacted cache to the authoritative one. This can be easily done by subtracting from $h_i^{(B)}$ and adding to $h_i^{(F)}$, the direct backend hits which are:

$$\begin{aligned} h_B(i|j=D)\cdot P(j=D) &= (1-e^{-A_{i,j=D}})\cdot P_i(j=D) \\ &= \frac{1-e^{-\frac{1}{K}\,p_i\,T_B(1+\eta_i)}}{1+\eta_i} \end{aligned} \tag{3.86}$$



(a) Single frontend - Hits vs. C_B

(b) Array of frontends - Hits vs. C_B

(c) Co-located frontends/backends - Hits vs. C_B

(d) Array of frontends - Hits vs. K'

Figure 3.7: Accuracy of two-layer caching models

3.5.4 Model validation and discussion

We validate the accuracy of our models by comparing their predicted cache hit ratio with values from numerical simulations and plot results in Fig. 3.7. We performed all experiments using a Zipf-distributed content popularity with 512,000 items and α = 0.8.

In Fig. 3.7a, Fig. 3.7b and Fig. 3.7c we show the accuracy of our models against various cache sizes for the single and array and co-located cases. In both cases we vary C_B between 2,000 and 64,000 items and maintain a constant ratio between backend and frontend cache sizes C_B/C_F equal to 8. We also maintain constant values of K and K' equal to 16. We deliberately selected small cache sizes because this is the most unfavourable condition for our model (see Theorem 3.6). Despite this, our model predicts performance very accurately under all cache sizes considered. Another important consideration is that all three deployment configurations, as long as each frontend cache and each backend cache have the same size yield practically identical cache hit ratio.

Finally, in Fig. 3.7d we further evaluate the accuracy of the array model maintaining constant values of C_B and C_F of 16,000 and 2,000 and varying the number of frontend caches K' from 1 to 64. Our model again predicts performance very well and, as we showed above, both frontend and backend cache hit ratio do not depend on the number of frontend caches.

3.6 Conclusion

In this chapter, we presented novel and practical results shedding light on the performance of sharded caching systems. Our analysis focused primarily on load balancing and caching performance aspects. With respect to load balancing, we characterised load imbalance among shards caused by content popularity skewness and heterogeneous item sizing. We also quantified how item chunking and the deployment of small frontend caches can reduce such imbalance. With respect to caching performance, we characterised the cache hit ratio of sharded systems both in isolation and deployed with various configurations of frontend caches. We believe that our result are easily applicable and can improve the design of such systems.

Chapter 4

Framework and algorithms for load-balanced content caching

4.1 Introduction

As already mentioned, two trends are expected to influence content distribution in both the short and long term. The first trend, with short term impact, is represented by the rapid growth of Network CDNs, *i.e.,* CDNs managed by network operators (see, for example [13], [53], [117]), which is driven by both technological and commercial incentives. The second trend, with longer term impact, is the increasing momentum gained by the emerging ICN paradigm, and in particular CCN [89] and NDN [183] architectures, within both the research and operator/vendor community.

Both trends are leading to a scenario in which network operators will be in charge of operating a networked caching infrastructure, which will likely be very complex, comprising several nodes and carrying a considerable amount of traffic. In a different fashion to traditional CDNs, whose main objective is to improve the performance of content distribution for their customers, network operators also have the additional objective of minimising the operating cost of their network. This includes reducing the amount of interdomain traffic crossing transit links and ensuring homogeneous utilisation of resources by evenly spreading traffic across network links and caching nodes. In addition, an operator has also knowledge of topological characteristics and can exert more control on network routing and cache placement, which it can use to better design and operate its distributed caching infrastructure.

In this chapter, we propose an integrated framework to achieve efficient, distributed, operator-driven caching that can be equally applied to both Network CDNs and the future ICN paradigm. More importantly, and differently from previous work, this framework provides a complete solution covering all aspects related to the deployment and operation of a caching system, *i.e., cache placement*, *content placement* and *request routing*.

Our solution is based on the use of sharding to distribute load across caching nodes and enable scalable request routing. As discussed in Chapter 3, sharding techniques have been extensively used to horizontally scale storage and caching systems for various applications. Sharding has also been used in the context of Web caches (where it has been more commonly referred to as *hash-routing*) to route requests among caching nodes co-located in the same physical facility (see, for example [100], [145],

[163]) but was never considered for geographically distributed environments, mainly amidst latency increase concerns.

In this chapter we revisit hash-routing techniques and demonstrate with analytical modelling and extensive simulations that they can be successfully applied to the case of ISP-wide caching deployments. While latency increase can be a concern when applying hash-routing techniques to distributed environments, we show that ISP-level caching is a sweet spot in which latency increase caused by request detouring is completely offset by the benefits of increased cache hit ratio. Hash-routing provides also additional benefits, such as better load balancing across network links and caching nodes, reduced interdomain traffic and better resilience against traffic spikes.

This chapter provides four main contributions. First, we design a complete caching framework extending hash-routing techniques to specifically target operator-managed content caching, where caches are geographically distributed. Second, we provide a thorough analytical modelling of our framework, covering cache hit ratio, latency and load balancing aspects. Third, we design a practical optimisation algorithm to compute the optimal cache placement in polynomial time. Fourth, we evaluate the performance of our framework with trace-driven simulations using real ISP topologies and real traffic traces.

Overall, we show that our proposed framework provides several advantages over similar state-of-the-art distributed caching techniques, including:

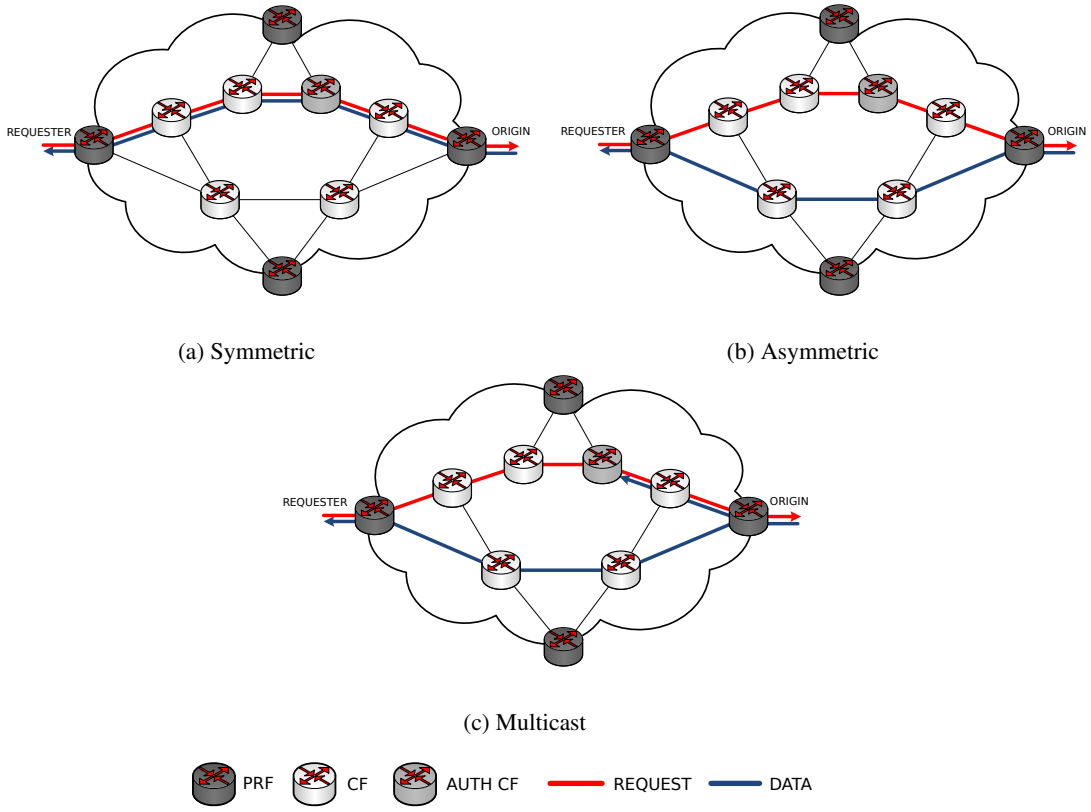
- Greater cache hit ratio, resulting in minimised inter-domain traffic and subsequent OPEX reduction.
- Inherent load-balancing across links and nodes.
- Robustness against localised hotspots and flash crowd events.
- Predictable performance, making the caching infrastructure easy to model and optimise.
- Tunable performance, giving operators greater flexibility.

The remainder of this chapter is organised as follows. In Sec. 4.2 we present the basic overall design of the proposed hash-routing schemes and investigate three extensions in Sec. 4.3. We then describe analytical models of the system in Sec. 4.4. Based on these results, we formulate a cache placement optimisation problem in Sec. 4.5. We then present an overall performance evaluation in Sec. 4.6. Finally, we summarise the contributions of this chapter and draw our conclusions in Sec. 4.8.

4.2 Overall framework design

Our proposed hash-routing framework comprises two types of functional entities: *Proxy and Routing Functions (PRF)* and *Caching Functions (CF)*. PRF and CF entities can be physically separated or co-located, but since they provide in principle separate functionalities we describe them separately.

Proxy and Routing Functions (PRF) are the first entities traversed by content requests when entering the cache network and they are in charge of forwarding them to relevant CF instances. They have knowledge of all CF instances in the network and their addresses (but do not need to know which content items are stored at each CF) and share a common hash function to map content identifiers to CF instances.



(a) Symmetric

(b) Asymmetric

(c) Multicast

Figure 4.1: Hash-routing schemes

Such a hash function does not need to be collision-resistant. Instead, it is desirable for its output to be produced incurring minimal processing overhead. Consistent hashing techniques [100] may also be used in order to minimise the number of content items being remapped as a result of CF addition or removal. Caching Functions (CF) are simply in charge of serving and storing the content items mapped to them by the shared hash function. PRFs can be physically deployed in access nodes with CFs in both access and core nodes or, in another deployment scenario, PRFs and CFs can co-exist in points of presence (PoPs).

The operation of the scheme is illustrated in Fig. 4.1a. When a requester issues a request for a content item, the request is forwarded to the closest PRF. The PRF identifies the CF responsible for that content by computing the hash function on the content identifier and forwards the request to that CF. If such CF has the requested content, it returns it to the user via the PRF. Otherwise, the request is forwarded towards the content origin through an egress PRF without looking up any other cache along the path. When the content item is returned by the content origin, the egress PRF forwards it to the authoritative cache that stores it and finally forwards it to the requesting user.

This solution can be easily implemented in both the Network CDN and ICN cases using common techniques. In the Network CDN case, user requests are forwarded to the closest PRF as a result of a DNS lookup, as commonly done in current CDNs [38]. Both PRFs and CFs operate as reverse HTTP proxies with persistent TCP connections running among them, which are used to forward HTTP requests among them. Differently, in the ICN case, user requests are forwarded to the closest PRF through name-based routing, which can be achieved by inserting appropriate forwarding entries in the FIBs of access ICN

routers. A PRF can then redirect a request to the authoritative CF by encapsulating the request packet with a header containing a name that identifies the authoritative cache. For example, if a request needs to be forwarded to CF A, the PRF can prepend a request header for name `/_cache/_a`. All ICN routers on the path are configured to route names with the `/_cache/_a` prefix to the location of CF A. When CF A receives the request packet, it removes the prepended header and processes the encapsulated packet. This is a standard approach, commonly used in ICN request routing to steer packets through instances of network functions [10]. There is no need for encapsulation to route content packets through the authoritative CF in case of cache miss. In fact, content packets are forwarded on the reverse path taken by request packets, and hence through the authoritative cache, using information stored in the PIT of intermediate routers. It should be noted that, since content packets are not modified, this technique does not affect the verification of content signatures.

The design of this hash-routing scheme exhibits two interesting properties. First, since a content item can be cached only by the CF instance resulting from the hash calculation, a specific content item can be cached at most in one node of the network. This maximises the number of distinct items cached within the ISP network and therefore reduces inter-domain traffic and load at origins. Second, localised spikes in traffic occurring in a network region are implicitly spread across links and caching nodes by the hash computation in a way similar in principle to Valiant Load Balancing (VLB) [172]. In Sec. 4.4.2 and 4.6.3 we show that our scheme can indeed achieve very good load balancing under common operational conditions.

As explained previously, a possible deployment model would consist in the co-location of PRFs and CFs at (a subset of) the operator's PoPs. PRFs could also be deployed separately from CFs. For example, they may be deployed in the access network (*e.g.,* in an EnodeB in the case of an LTE network or in a DSLAM or BRAS in an xDSL fixed broadband access network). In this case, the PRFs deployed at the edge may also be equipped with a small caching space which is operated autonomously, *i.e.,* it can cache any content. We elaborate on this aspect in Sec. 4.3.2.

We model this framework analytically in Sec. 4.4. In the following section we present extensions to this base design that support additional desirable properties and provide knobs to fine-tune the performance. It is worth noting that, by nature, these extensions add some complexity to the overall design and cannot, as such, be analytically modelled. Therefore, their performance is only evaluated with simulations.

4.3 Extensions

4.3.1 Asymmetric and multicast content routing

In the base hash-routing scheme described above, request and content paths are symmetric, *i.e.,* in case of a cache miss, the content (on its way back from the origin) is forwarded first to the cache and then to the requester, following the reverse path of the request. While this routing scheme is simple to implement and manage, the path stretch resulting from request and content routing through off-path caching nodes may lead to increased latency and link load.

To address this limitation, we propose two alternative content routing schemes in addition to the

base one proposed above, which we refer to as *Symmetric* (see Fig. 4.1a). These schemes, that we named, respectively, *Asymmetric* and *Multicast*, differ from the symmetric scheme only with respect to the delivery of contents entering the network as a result of a cache miss. The asymmetric routing scheme (Fig. 4.1b) always routes contents through the shortest path. The content is cached in the responsible cache only if it happens to be on the shortest path between the origin and the requester, otherwise it is not cached at all. In the multicast routing scheme (see Fig. 4.1c), when the egress PRF receives a content, it multicast it to both the authoritative cache and the requester. If the authoritative cache is on the shortest path between content origin and requester, then symmetric, asymmetric and multicast schemes operate identically.

All three content routing options present both advantages and disadvantages. The choice of content routing however gives network operators a knob to tune performance.

Both multicast and asymmetric routing reduce user-perceived latency because contents fetched from the origin are delivered to the requesting user without any detour. However, they may not be applicable to future Internet architectures where only symmetric request/response paths are allowed, *e.g.,* CCN and NDN. This is however likely to change in the near future, since PARC (currently leading the CCN architecture development) is considering supporting asymmetric paths.

In addition, asymmetric routing reduces link load since contents (which are much larger than requests) are always delivered though the shortest path. However, CF instances are populated with contents entering the domain only if they happen to be on the shortest path from origin to requester. This would not be a problem if content popularity varies slowly. On the contrary, it would actually increase cache hit ratio since it would reduce the impact of one-timers. If, differently, request patterns exhibit strong temporal locality, a content item may need to be requested several times before it is actually cached (especially if the responsible cache is located on an underutilised path), hence affecting caching performance.

Multicast routing achieves the same latency of asymmetric routing and only requires one content request for a content to be cached, as in symmetric routing. However, since at each content request, a content item must be forwarded to the authoritative cache, it leads to greater link loads than the one achieved by asymmetric routing, especially when cache and requester paths have a limited overlap.

4.3.2 Hybrid caching

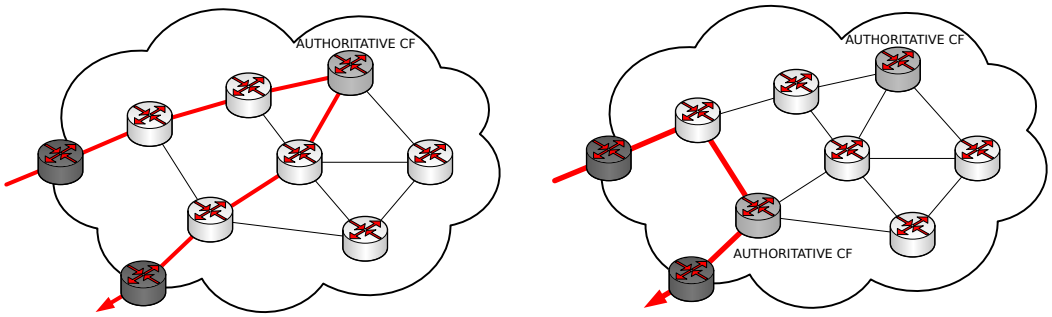
Another proposed extension to the base design consists in allocating part of the overall caching space to operate autonomously, *i.e.,* cache any content indiscriminately, regardless of the content-cache mapping. This could be implemented in two different ways, depending whether CF and PRF entities are co-located or not. If PRF and CF are co-located, this extension can be implemented by allocating a static fraction of the caching space at CF nodes to cache any content for which the CF is not responsible. Differently, in case PRF and CF are not co-located, a small caching space can be deployed within the PRF and CF entities can be used to only cache content items for which they are responsible.

This can provide two main advantages. First, it allows a small number of very popular contents to be replicated in multiple nodes, instead of just one, hence possibly reducing overall latency and link load. Second, as extensively discussed in Sec. 3.3.5, a small frontend cache achieves better load balancing

across backend caching nodes. In fact, while base hash-routing evens out localised traffic hotspots by spreading traffic originated from each requester across caches, any peak in demand for a specific content item will always be served by the same cache. By caching a small fraction of very popular contents in multiple caching nodes, the system is made robust to variations in content popularity. This also makes the system more robust to adversarial workloads targeted at overloading a specific caching node, as shown by Fan *et al.* [57]. Additionally, in case asymmetric content routing is adopted, this extension ensures that a requested content is cached the first time it is requested even if the responsible cache is not on the shortest path between the origin and the requester.

Performance can be tuned by selecting what fraction of caching space to dedicate to uncoordinated caching. Increasing the uncoordinated caching space leads to greater robustness towards flash-crowd events and lower latency in accessing the most popular contents. However, it reduces the number of distinct items cacheable in the network, hence affecting the overall cache hit ratio (possibly leading to a greater overall latency) and robustness against localised spikes in traffic. It should be noted that utilising uncoordinated caching space makes the system behave as a hybrid between edge caching and base hash-routing. At one extreme, where all caching space is operated in a coordinated manner, the system is a base hash-routing one. At the other extreme, the system behaves as a pure edge caching system.

4.3.3 Multiple content replication



(a) Single replica (b) Multiple replicas

Figure 4.2: Single and multiple content replication

Base hash-routing design allows a content item to be cached at most once in a network domain, hence maximising the efficiency of caching space. However, in case of very large topologies, this may result in high path stretch, possibly leading to high latency (see, for example, Fig. 4.2a).

To address this issue, we propose an extension where the content-to-cache hash function maps to k distinct nodes instead of just one. As a result, multiple nodes (ideally well distributed over the network) are responsible for each content (see Fig. 4.2b). This ensures that the worst case path stretch to reach a responsible cache is reduced.

When a PRF processes a content request and computes the content hash, it resolves the content to k distinct caching nodes. The PRF forwards the request to the closest of the k CFs responsible for the

content. In case of cache miss, two approaches can be adopted:

1. The CF forwards the request directly to the content origin.
2. The CF forwards the request towards the original source through (all or a subset of) the other CFs responsible for the content, if this can be done with limited path stretch.

More sophisticated approaches are also possible. For example, it could be possible for PRF instances to dynamically select which authoritative CF to redirect the request to, based for example on the current load. We reserve this investigation for future work.

Similarly, when a content enters the network after an origin fetch, several options can be adopted. The content can be forwarded symmetrically, asymmetrically or with multicast. Also, in case of symmetric content routing, if several caches are looked up, contents can be placed using various on-path meta algorithms such as LCE [114], LCD [113] or ProbCache [139].

The degree of replication k enables the fine-tuning of performance by trading off latency with cache hit ratio. Hash-routing operating with multiple content replicas is a hybrid between hash-routing and on-path caching meta algorithms. At one extreme ($k = 1$) the system behaves as pure hash-routing scheme. At the other extreme (k equal to the number of caching nodes) the system behaves exactly as the on-path meta algorithms used for content routing, or as edge caching in case requests are forwarded to the content origin in case of a first miss.

4.4 System modelling

This section presents a thorough modelling of the base hash-routing system presented in Sec. 4.2. We start by investigating the overall cache hit ratio of a network of caches operating under a symmetric hash-routing scheme and the load balance across caching nodes. This analysis applies results derived in Chapter 3. We then use these results to derive closed-form equations for the average content retrieval latency.

Consistently with previous work and with the rest of this thesis, we assume in our analysis that contents are requested according to the Independent Reference Model (IRM).

In the following, we describe the notation adopted in the remainder of the chapter, with reference to the abstract representation of our system depicted in Fig. 4.3. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed graph with vertices $\mathcal{V}$ and edges $\mathcal{E}$ representing the operator's network and let $\mathcal{O}$ be the set of contents items. A set of caches $\mathcal{C} \subseteq \mathcal{V}$ are deployed at a subset of network nodes. Each cache is capable of storing $S < \lfloor |\mathcal{O}|/|\mathcal{C}| \rfloor$ items. We assume that all caches have the same capacity and operate according to the same replacement policy and that this replacement policy is defined by characteristic time (see Definition 3.2). Content items are mapped uniformly to caches according to a hash function $f_H : \mathcal{O} \rightarrow \mathcal{C}$. Each node $v \in \mathcal{V}$ issues requests for content $o \in \mathcal{O}$ with rate $\lambda_o^{(v)}$. The cumulative rate of requests originated at node v is $\lambda^{(v)} = \sum_{o \in \mathcal{O}} \lambda_o^{(v)}$. The network-wide request rate is $\Lambda = \sum_{v \in \mathcal{V}} \lambda^{(v)}$. We further assume that each content is permanently stored at only one location outside the network and denote $\mathcal{O}_s$ as the set of contents stored at origin node s. Finally we denote $\mu^{(s)}$ as the rate of miss requests forwarded towards the content origin via egress node s.

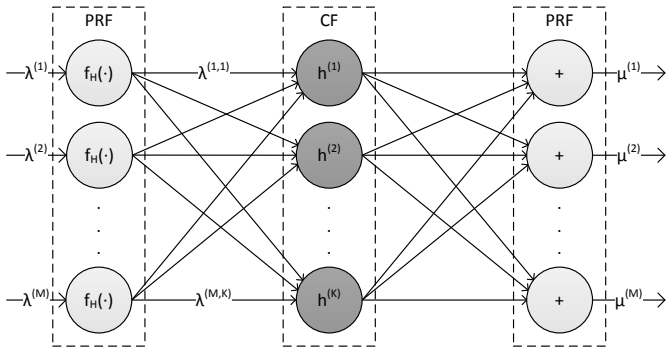


Figure 4.3: Model of a network of hash-routed caches

4.4.1 Cache hit ratio

We start by investigating the cache hit ratio performance of a base hash-routing system. We demonstrate that, under the operational conditions of our interest, *the overall cache hit ratio of the network is approximately equal to the cache hit ratio of a single cache with size equal to the cumulative cache size of the network*.

We draw this conclusion by applying the results of Theorem 3.6. This theorem is applicable as long as three conditions are satisfied:

1. All caches have equal size.
2. All caches operated under the same replacement policy, which is determined by characteristic time.
3. The system is subject to IRM demand.

Conditions 1 and 2 are satisfied by definition. Condition 3 is also satisfied because the demand entering the system of caches is simply the sum of demands at each ingress node, which, by definition, are IRM and the sum of IRM demands is also IRM.

Theorem 3.6 shows that if each cache is large enough then it is possible to use a mean field approximation to estimate the characteristic time with limited impact on results accuracy. In particular:

$$P\left(\left|\tilde{S}-S\right| \geq \epsilon S\right) \leq \left(1-\frac{1}{|\mathcal{C}|}\right)\frac{1}{\epsilon^2 S} \tag{4.1}$$

where $\tilde{S}$ is the size of a single cache associated to the approximated characteristic time.

Even assuming the unfavourable case of $|\mathcal{C}|\rightarrow +\infty$ and $S = 10^6$, the probability that the mean field approximation induces an error above $0.01\,S$ is 1%. We analysed the top 300K Web sites of the Alexa index [1] and we observed that, as of September 2015, the average size of a Web object is 24 KB. Even deploying caches on dated machines with only 24 GB of DRAM reserved for content cache and no SSD or HDD, caches would be able to hold on average at least 10^6 content objects each, which, as we showed above, is a figure for which the approximation yields an error no greater $0.01\,S$.

We can therefore conclude that in the operational conditions of our interest, we can accurately model the system of hash-routed caches as a single cache with size equal to the cumulative caching size of the system.

[1] We collected this information by querying the HTTP archive dataset [81].

4.4.2 Load balancing

We analyse how uniformly the hashing of contents to caches spreads requests across caching nodes, in spite of number of caching nodes and skews in content popularity distribution.

For this analysis, we further assume that content popularity follows a Zipf distribution with skewness parameters α. We apply the results of Theorem 3.2, which states that the coefficient of variation of the fraction of requests assigned to each node $c_v(L)$ corresponds to:

$$c_v(L) \approx \begin{cases} \sqrt{|\mathcal{C}|-1}\sqrt{\frac{(1-\alpha)^2[(|\mathcal{O}|+1)^{1-2\alpha}-1]}{(1-2\alpha)[(|\mathcal{O}|+1)^{1-\alpha}-1]^2}}, & \alpha \notin \{\frac{1}{2}, 1\} \\ \frac{\sqrt{(|\mathcal{C}|-1)\log(|\mathcal{O}|+1)}}{2(\sqrt{|\mathcal{O}|+1}-1)} & \text{if } \alpha = \frac{1}{2} \\ \sqrt{\frac{|\mathcal{O}|(|\mathcal{C}|-1)}{(|\mathcal{O}|+1)\log^2(|\mathcal{O}|+1)}} & \text{if } \alpha = 1 \end{cases} \tag{4.2}$$

Analysing Eq. 4.2, we can observe that load imbalance increases proportionally to α and $|\mathcal{C}|$ and decreases proportionally to $|\mathcal{O}|$. Even though load imbalance increases superlinearly as $\alpha \to 1$, it remains fairly low as long as the content population is very large, which is the case for Internet traffic. For example, even in the unfavourable case of $\alpha \to 1$, $|\mathcal{O}| = 10^9$ and $|\mathcal{C}| = 128$ we have $c_v(L) = 0.54$. In addition, as shown in Sec. 3.3.4, in the specific case of chunk-level caches where the hashing is performed on chunk identifier (as opposed to the identifier of the content object) load imbalance is reduced by a factor of $\sqrt{M}$ where M is the number of chunks in which content objects are partitioned. This is for example the case of packet-level caches envisaged by ICN proposals.

For all these reasons, we can reasonably assume that at the operating conditions of our interest, content requests are uniformly spread across caches. The goodness of this assumption, as well as of the assumptions made in Sec. 4.4.1 are validated by the accuracy of the latency model presented later, which heavily relies on these assumptions.

4.4.3 Latency

One advantage of hash-routing over prior work is its performance predictability, that makes it considerably simpler to model and to optimise than other schemes. In this section, we build on the results of Sec. 4.4.1 and 4.4.2 and provide closed-form equations to compute user-perceived latency in a network operating under a base hash-routing scheme.

We denote the system-wide cache hit ratio of content $o \in \mathcal{O}$ as h_o, which, as showed above, can be estimated using known techniques assuming modelling the system of caches as a single large cache. We then denote as δ_e the average latency of link $e \in \mathcal{E}$ and denote as $\mathcal{P}_{s,t} \subseteq \mathcal{E}$ the network path from node s to node t, *i.e.,* the set of edges traversed from s to t. We further denote as $\delta_{s,t} = \sum_{e \in \mathcal{P}_{s,t}} \delta_e$ the average latency of path (s, t). We assume that the latency does not depend on the size of the message, *i.e.,* it is dominated by queueing delay and propagation delay as opposed to transmission delay.

Assuming that each cache in the network has the same cache hit ratio (Sec. 4.4.1) and that load is equally spread across caching nodes (Sec. 4.4.2), we can compute the cumulative rate of requests $\mu^{(s)}$

served by origin s as:

$$\mu^{(s)} = \sum_{v \in \mathcal{V}} \sum_{o \in \mathcal{O}_s} \lambda_o^{(v)} (1 - h_o) \tag{4.3}$$

We can now determine the average latency perceived by a user to retrieve a content item. This value corresponds, in case of cache hit, to the round-trip time between the requester r and the responsible cache c, denoted as $D_{r,c}$. In the case of cache miss, latency is increased by the round-trip time between cache c and content origin s, denoted as $D_{c,s}$. Each latency component is equal to:

$$D_{r,c} = \frac{1}{\Lambda |\mathcal{C}|} \sum_{r \in \mathcal{V}} \lambda^{(r)} \sum_{c \in \mathcal{C}} (\delta_{r,c} + \delta_{c,r}) \tag{4.4}$$

$$D_{c,s} = \frac{1}{\Lambda |\mathcal{C}|} \sum_{s \in \mathcal{V}} \mu^{(s)} \sum_{c \in \mathcal{C}} (\delta_{c,s} + \delta_{s,c}) \tag{4.5}$$

$$D = D_{r,c} + D_{c,s} \tag{4.6}$$

To demonstrate the goodness of this approximation we show in Fig. 4.4 the latency measured from simulations and compare it with the results of our model. The graph shows an excellent agreement between model and simulation results. This proves not only the accuracy of our latency model, but also the goodness of our assumptions about cache hit ratio and load balancing upon which this model is based.

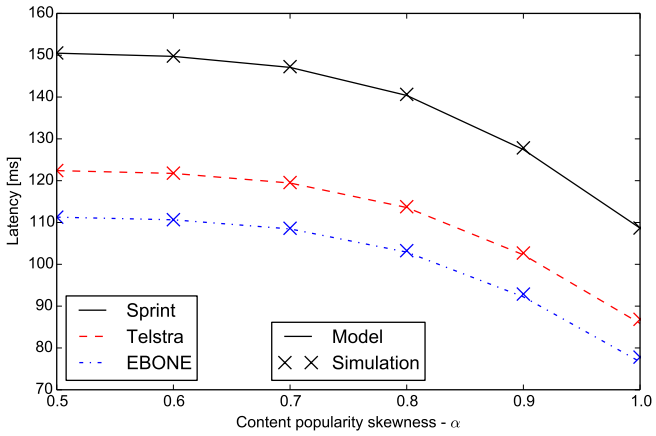


Figure 4.4: Accuracy of latency model

4.5 Optimal cache placement

In this section, we provide a simple polynomial-time algorithm to compute the optimal placement of caching nodes across a network given the target number of caching locations p, the size of a single caching node S and the global demand. This algorithm is loosely based on the greedy uncapacitated placement algorithm proposed in [40]. Its objective is the minimisation of the overall latency assuming equal cache hit ratio among caches (see Sec. 4.4.1) and uniform load balancing among caching nodes (see Sec. 4.4.2).

The algorithm comprises the following three steps.

1. Compute the global cache hit ratio h, *i.e.,* the cache hit ratio of a cache with size pS. It can be either computed numerically or by other means such as the well known Che's approximation [35], [126].

2. Compute the cost d_v of each cache candidate node v. This cost corresponds to the average latency that the traffic would experience if node v was the only cache in the system, of size pS. Using the notation adopted in the rest of the paper, this corresponds to:

$$d_v = \sum_{u \in \mathcal{V}} \left(\lambda^{(u)} \delta_{u,v} + \mu^{(u)} \delta_{v,u} \right) \tag{4.7}$$

3. Place caches at the p nodes with lowest cost d.

The proof of optimality is straightforward. In fact, as per the assumptions laid out above, the overall latency of the system D can be computed as:

$$D = \frac{1}{p} \sum_{v \in \mathcal{V}} d_v X_v \tag{4.8}$$

where variable X_v takes value 1 if a cache is deployed in node v and 0 otherwise. It is obvious that the solution that minimises D satisfying the constraint $\sum_{v \in \mathcal{V}} X_v = p$ consists in setting $X_v = 1$ for the p nodes with the lowest cost d_v and $X_v = 0$ for the remaining nodes.

The number of caching nodes to deploy p is an input of the optimisation problem, which could be selected according to a number of criteria. A simple criteria could be to select the smallest p that guarantees that each caching node is able to handle the expected peak load. In this case, if R is the overall peak rate of requests expected by the system and r is the maximum rate of requests that a single caching node could handle, then we could select $p = \lceil R/r \rceil$.

It should be noted that in our problem formulation we do not account for link capacity constraints. The rationale for this decision is that in normal operational conditions, intradomain network links operate at a small fraction of their capacity, as we learned from discussions with a large European ISP operating a CDN infrastructure. Links operate at high load conditions only in the case of transient traffic spikes, which are implicitly addressed by our framework. We validate the robustness of our solution in terms of link load balancing in Sec. 4.6.3.

4.6 Performance evaluation

In this section we present the results of our performance evaluation. We investigate the performance of the hash-routing schemes described above and compare them against previously proposed techniques under a variety of operational conditions. We also present how each proposed hash-routing extension helps fine-tuning performance.

4.6.1 Setup and methodology

We evaluate the performance of our proposed schemes with trace-driven simulations, using the *Icarus* simulator (see Appendix B).

We performed our analysis using various combinations of real traffic traces and topologies.

Table 4.1: Network topologies

ASN	Name	Region	# PoPs	# backbone links
1221	Telstra	Australia	104	150
1239	Sprintlink	US	315	971
1755	EBONE	Europe	87	160
3257	Tiscali	Europe	161	327
3967	Exodus	US	79	146
6461	Abovenet	US	138	371

We used six PoP-level ISP topologies from the RocketFuel dataset [157] annotated with inferred link weights and latencies (see Tab. 4.1). These six topologies provide a good variety of sizes (from 79 to 315 PoPs) and refer to different geographic regions (US, Europe and Australia). We generate content requests from clients attached to each PoP with equal rate and assume that all content origins are outside the ISP. We set the cumulative cache size of the ISP to be equal to 0.1% of the content catalogue size. Each caching node is equipped with the same cache size and replaces contents according to the LRU policy.

We used workloads derived from three real traces (see Tab. 4.2). For each of these traces, we used the first 25% of requests to warm up caches and measured performance over the remaining 75% of requests. The first workload we used is a 1-day trace containing a sample of 10% of all requests received by Wikipedia [171]. The other two workloads are from the IRCache dataset [87], which contains all HTTP requests collected by a set of proxies deployed at a number of universities in the US. We used two traces from this dataset. The first, which we labelled *IRCache ALL*, is a 2-day trace collecting all requests coming from seven different sites. The second, labelled *IRCache UIUC* is a 7-day trace containing only requests coming from the proxy deployed at the University of Illinois Urbana-Champaign.

Table 4.2: Traffic traces

Trace	# requests	# items	Zipf α	Duration	# requests per day
Wikipedia	11,566,029	1,834,747	0.99	1d	11,566,029
IRCache (ALL)	8,278,100	5,240,029	0.70	2d	4,139,050
IRCache (UIUC)	5,925,463	3,964,700	0.70	1w	846,494

The traces used for our experiments have different characteristics. The Wikipedia trace has a more skewed content popularity distribution than IRCache ones, as evidenced by the greater value of Zipf α parameter (0.99 vs. 0.70). All traces have a similar amount of requests (6M, 8M and 11M) but over different time periods (1, 2 and 7 days). As a result, they have a different amount of requests per day that leads to different temporal locality characteristics. Wikipedia is the most stationary while IRCache UIUC is the most dynamic. Finally, the two IRCache traces, despite having similar content popularity skeweness, cover a different geographic area. IRCache ALL encompasses requests from seven different

regions, while the IRCache UIUC only from one single campus. We believe this mix of traces covers a large variety of realistic operational conditions.

We report that for each experiment requiring randomised configuration (*i.e.,* assignment of contents to origin nodes), we repeated each experiment 10 times and computed the 95% confidence interval using the repetitions method. We omitted to plot error bars if too small to be distinguishable from point markers. Finally, for the sake of conciseness, we present results for all 18 topology-workload combinations at our disposal in Sec. 4.6.2 only. In all other cases, we report that we observed similar results among all combinations. Therefore we only present results for the Wikipedia workload, which we selected because this dataset has the largest number of requests. Instead, the Telstra, Sprint and EBONE topologies were chosen because they cover three different geographic areas (Australia, US and Europe) and also have different sizes.

4.6.2 Base hash-routing

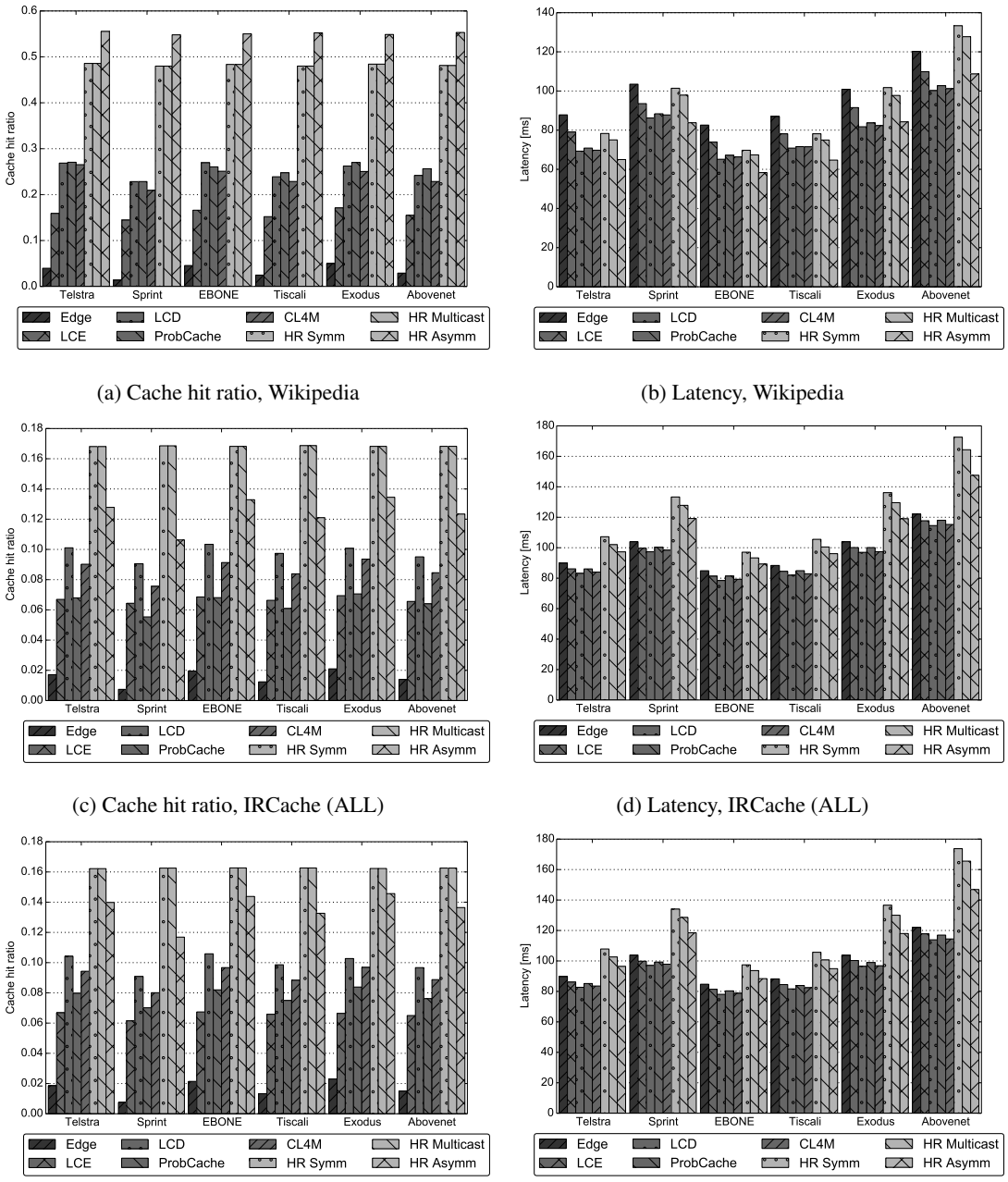
We begin by analysing the performance of the three content routing schemes proposed (symmetric, asymmetric and multicast) and compare them against other content placement strategies (namely edge caching, LCE [114], LCD [113], ProbCache [139] and CL4M [33]) across various topologies and workloads. Since there are no practical optimised cache allocation algorithms for most of the content placement techniques that we use as baseline, we present the results referring to a dense cache deployment (a cache for each PoP). The results presented here are simply a lower bound of hash-routing performance, since, as we show later, optimisations and extensions further enhance performance.

Fig. 4.5 shows that hash-routing schemes achieve a considerably higher cache hit ratio than other caching schemes, across all topologies and workloads considered. This is expected since hash-routing, differently from other schemes, maximises the number of distinct content items cached in the network. Symmetric and multicast routing achieve identical cache hit ratio by design, since in both cases each cache miss results in the insertion of the requested content in the responsible cache.

On the other hand, asymmetric routing achieves better cache hit ratio than symmetric/multicast routing in the case of Wikipedia workload and worse in the case of IRCache workloads. This difference is explained by the fact that, as shown in Tab. 4.2, IRCache workloads have a lower temporal density of requests and therefore exhibit a more dynamic behviour than the Wikipedia workload. With asymmetric routing, contents are inserted in cache only for a fraction of cache misses, as if each caching node operated with probabilistic insertion. This is known to reduce the impact of unpopular contents on LRU caches leading to a higher steady-state cache hit ratio but slower convergence. This justifies the better performance with more stationary workloads and worse performance with more dynamic ones.

With respect to latency instead, we notice that the performance achieved by hash-routing schemes are pretty similar to those achieved by on-path schemes. Under all workloads and topologies considered, symmetric routing achieves a greater latency than multicast and asymmetric schemes that always forward content items over the shortest path to the requester.

The main conclusion is that *hash-routing algorithms, even in an unoptimised dense deployment achieve a considerable cache hit ratio (which also leads to reduced OPEX as a result of lower interdomain*



(a) Cache hit ratio, Wikipedia

(b) Latency, Wikipedia

(c) Cache hit ratio, IRCache (ALL)

(d) Latency, IRCache (ALL)

(e) Cache hit ratio, IRCache (UIUC)

(f) Latency, IRCache (UIUC)

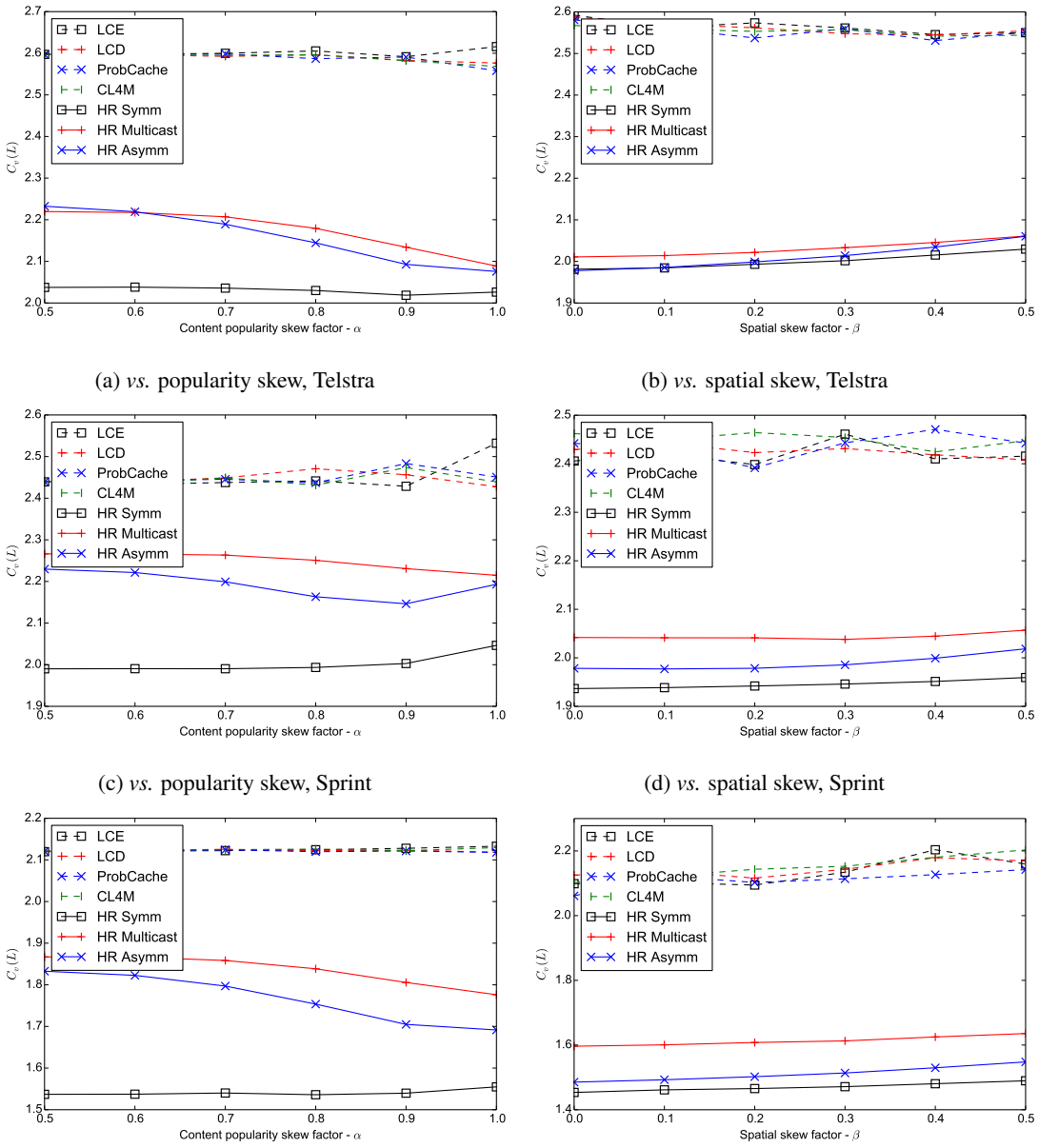
Figure 4.5: Cache hit ratio and latency of base hash-routing schemes

traffic) while still yielding latency values similar to other schemes.

4.6.3 Load balancing

In this section, we evaluate the load balancing properties of the proposed hash-routing schemes, which are designed to balance load implicitly by uniformly spreading traffic over caching nodes similarly in principle to Valiant Load Balancing (VLB) [172].

We already investigated the performance of load balancing across caching nodes in Sec. 4.4.2. Here we focus on the balancing of load across network links, which could not be evaluated analytically. We



(a) *vs.* popularity skew, Telstra

(b) *vs.* spatial skew, Telstra

(c) *vs.* popularity skew, Sprint

(d) *vs.* spatial skew, Sprint

(e) *vs.* popularity skew, EBONE

(f) *vs.* spatial skew, EBONE

Figure 4.6: Load balancing across links *vs.* popularity and spatial skew

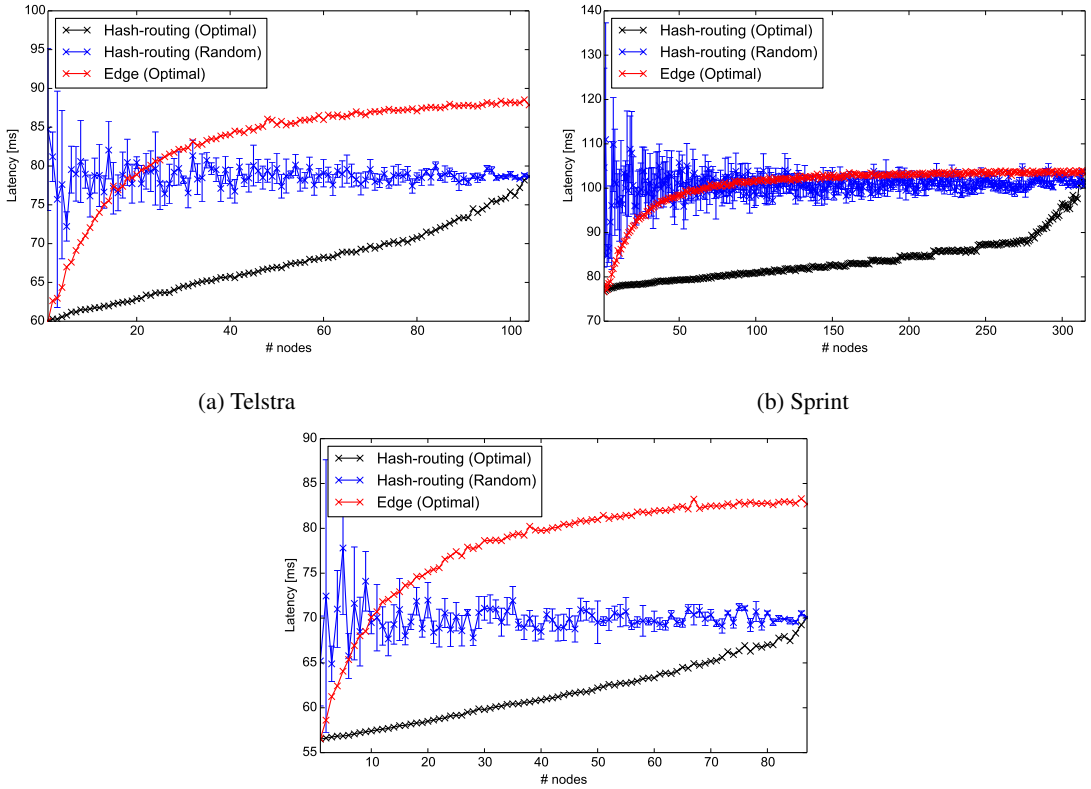
quantify it using the coefficient of variation of link load across all links of the network. In addition to analysing how well traffic is balanced across links, we also investigate robustness against variations in content popularity and geographic distribution of requests.

To investigate load balancing against variations in content popularity, we run experiments in which each PoP generates requests with equal rates and content popularity following a Zipf distribution with coefficient α that we vary between 0.5 and 1.0. The greater the value of α, the more skewed the content popularity distribution. Differently, to investigate sensitivity against geographic heterogeneity in content request rates (and hence against the presence of request hotspots), we run experiments using the Wikipedia workload but with the rate of requests generated by each PoP not uniformly distributed, as in the case

above. On the contrary, rates are assigned to PoPs following a Zipf distribution with coefficient β that we vary between 0 and 0.5. If $\beta = 0$, all PoPs originate requests with equal rates. Incrementing the value of β increases the difference between request rates across PoPs.

In Fig. 4.6, we present the results focusing on three RocketFuel topologies. From the graphs it can be immediately noticed that all three hash-routing schemes achieve a considerably better load balance (*i.e.,* lower coefficient of variation) than all other caching schemes for all topologies and all values of α and β considered. Among hash-routing schemes, symmetric content routing yields the lowest load imbalance. This is justified by the fact that, differently from other schemes, in case of cache miss all contents are always delivered through the responsible cache, hence leading to a greater scattering of traffic across network links. Finally, it is also worth noting that hash-routing plot lines are mostly horizontal, meaning that load imbalance does not vary significantly as α and β change. This shows that hash-routing load balancing, in addition to being better than other schemes, is also robust against variations in both content popularity and geographic distribution of requests.

4.6.4 Optimal placement and sparse deployment



(a) Telstra

(b) Sprint

(c) EBONE

Figure 4.7: Performance of cache placement algorithms

In this section we show the latency improvement provided by an optimal placement of caches according to the algorithm presented in Sec. 4.5 as opposed to a random placement.

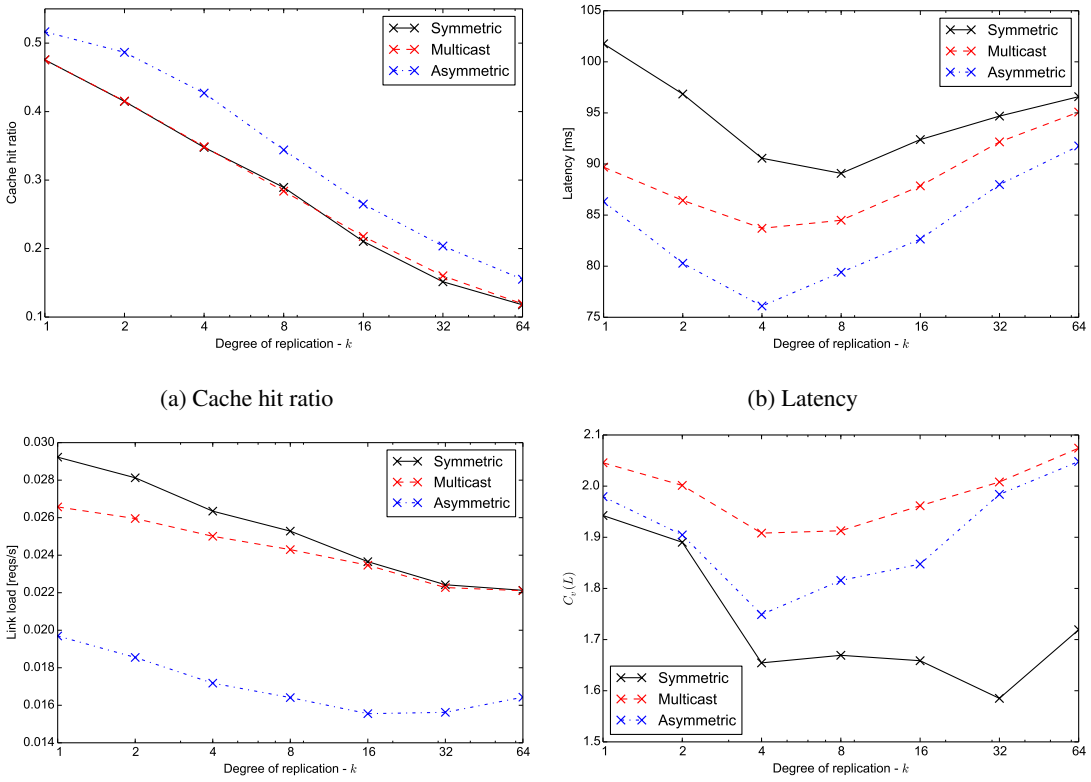
To add a further element of comparison, we also analyse the latency achieved by edge caching with

optimal cache placement. In this case the optimal placement can be mapped to a p-median location-allocation. We solve it using the Adjusted Vertex Substitution (AVS) algorithm [11], which is a recently proposed improvement of the well-known vertex substitution/interchange algorithm [162].

Results, depicted in Fig. 4.7, show first of all that optimal cache placement strongly reduces latency for all topologies and number of caching nodes considered. When caches are deployed over few nodes, the latency yielded by random placement strongly depends on the specific realisation, as shown by the large error bars, which represent standard deviation across realisations. As the number of caching nodes increases, latency variability decreases, as the number of possible combinations diminishes and converges to the performance of optimal placement when every PoP has a cache.

In addition, these experiments show that optimised hash-routing achieves considerably lower latency than optimised edge caching for all topologies and number of nodes considered. This in particular demonstrates that hash-routing is a good solution for both sparse and dense cache deployments.

4.6.5 Multiple replication



(a) Cache hit ratio

(b) Latency

(c) Link load

(d) Link load imbalance

Figure 4.8: Performance of multiple replication, Sprint topology

The path stretch incurred by hash-routing schemes for routing traffic through off-path caches may impact latency, especially in very large topologies. Despite results depicted in Fig. 4.5 show that even in large topologies hash-routing latencies are comparable to those achieved by other caching schemes, it would be beneficial to have knobs allowing operators to further reduce latency, possibly trading off cache hit ratio. For this reason, we proposed in Sec. 4.3.3 a design extension that enables multiple caching

nodes to be responsible for each content. In this section, we evaluate the performance of hash-routing schemes with multiple content replication and show that by replicating contents more than once it is in fact possible to improve latency by trading cache hit ratio.

The problem of efficiently assigning contents to caches with multiple replicas is considerably more complex than the single replica case. For practical purposes, we propose a simple placement algorithm to allocate content replicas. First, we set the degree of replication k (*i.e.,* the number of caching nodes responsible for each content). Then we group nodes in clusters in such a way that the latency among nodes of the same cluster is minimised. We do so by applying the Partitioning Around Medoids (PAM) clustering algorithm [103]. Then we map contents to each cluster independently, so that in each cluster there is one cache responsible for each content and, therefore, in the entire network there are k caches responsible for each content, one per cluster. It should however be noted that this algorithm does not ensure that each cluster has the same or even a comparable number of nodes.

We experimented applying this placement to the largest topology of the RocketFuel dataset (*i.e.,* Sprint) and testing it with the Wikipedia workload. We investigated two routing options: (i) querying all responsible caches on the cluster-level shortest path from requester to origin and (ii) querying only the responsible cache belonging to the same cluster of the requester and, in case of a miss, route the request to the origin without querying any other cache.

We have observed that the first option generally yields poor results as it provides only a marginal reduction in path stretch, not sufficient to compensate the loss in cache hit ratio. As a consequence, increasing the number of clusters leads to both a reduction of cache hits and an increase of latency and link load. Nevertheless, we do not rule out the possibility that a more sophisticated content-to-cache assignment algorithm could provide better results. We reserve this analysis for future work.

Differently, the second option provides very interesting results, which we plotted in Fig. 4.8. As shown in the figure, increasing the number of replicas reduces the cache hit ratio. This is expected since the number of distinct content items that the domain can cache diminishes. An interesting aspect is that symmetric and multicast content routing do not necessarily yield identical cache hit ratios if $k > 1$. This occurs because, in case of a cache miss, contents are inserted in the responsible cache of the cluster to which the requester belong as well as responsible caches on other clusters if they occur to be on the delivery path. This opportunistic insertion in caches of other clusters depends on the content path, which differs between symmetric and multicast routing.

Regarding latency, we identify a local minimum corresponding to $k = 4$ for asymmetric and multicast and $k = 8$ for symmetric content routing. Further increasing the number of clusters leads to worse latency because decreasing cache hit ratio is no longer offset by a substantial reduction in path stretch.

Link load also decreases by increasing the value of k. However, differently from the case of latency, there is a local minimum only for asymmetric routing (for $k = 16$), while for symmetric and multicast routing, link load always decreases as the number of clusters increases.

Finally, we observe that link load imbalance (quantified by the coefficient of variation of link load)

is not adversely impacted by multiple content replication. On the contrary, a modest degree of replication even reduces load imbalance.

The main conclusion is that *multiple replication of contents is effective in providing a knob to easily trade off cache hit ratio with latency*. A small degree of replication is sufficient to achieve a substantial latency reduction even using simple placement algorithms such as the k-medoid clustering we proposed.

4.7 Related work

We already provided an extensive overview of previous work regarding content caching in Chapter 2.

It is however worthwhile discussing two further streams of work related to the contributions of this chapter. These concern hash-routing protocols for content caching and caching systems designed to optimise operator-wide performance metrics.

4.7.1 Hash-routing

Hash-routing is a well known technique widely used in Web caches albeit only in the context of co-located caches. The classic approach would consist in using modulo hashing on a content identifier to map a content to a server. This approach however has the major drawback that the addition or removal of a server causes a considerable remapping of contents among servers, leading to a transient drop of performance. This problem was originally addressed by Thaler and Ravishankar [163], who proposed a mapping algorithm named Highest Random Weight (HRW). According to HRW, each content is mapped to an ordered list of servers. Requests are routed to the first server of the list and, in case of high load or failure, to the next server of the list until an available server is found. This approach has then been adopted in CARP [173].

A major breakthrough came with the invention of consistent hashing [100], which guarantees that in case of addition or removal of a servers only the contents mapped to the added/removed server are remapped.

Other noteworthy contributions on hash-routing come from Ross [145], who investigated the performance of hash-routing protocols in comparison to other cooperative caching schemes and Tang and Chanson [161] who proposed adaptive load balancing techniques for hash-routing by tuning hashing weights.

All these hashing techniques can be applied to our proposed framework as well, since it does not mandate any specific hashing scheme.

Subsequent to the publication of the contributions presented in this chapter, further proposals have investigated the application of hash-routing techniques in the context of ICN. Among those there is CoRC [40], which proposes to use hash-routing techniques among ICN routers of a domain like our framework, but with main objective of reducing the size of a forwarding table that each router needs to store by partitioning it across routers. Also Saha *et al.* [150] proposed the use of hash-routing for cooperative caching but focusing on the specific use case of inter-AS caching without taking intra-AS design considerations into account, such as load balancing and optimisation aspects, which our proposal addresses. Therefore their approach is complementary to ours which focuses on intra-AS caching.

4.7.2 Distributed caching to optimise operator-wide performance metrics

Despite considerable research effort has focused on the design of cooperative caching schemes, most work focuses on the general case of caches not necessarily managed by an ISP. Our approach, by specifically addressing the case of operator-managed content caching, enables the optimisation of ISP-wide performance metrics. Two recent pieces of work also addressed the case of optimising operator-wide metrics, albeit providing solutions completely different from ours. Pacifici and Dan [136] investigate the case of content-level peering, in which ASes cooperatively cache contents in their networks. Using game-theoretical concepts, they propose caching algorithms enabling a stable cache allocation. Araldo *et al.* [9] propose an on-path meta-caching policy driven by economic cost of retrieving content items from provider ISPs.

4.8 Conclusion

In this chapter we presented hash-routing techniques for use in geographically distributed environments. We showed that these hash-routing schemes provide several advantages in comparison to state-of-the-art techniques.

They provide excellent cache hit ratio, hence reducing transit costs. They also contribute to reduce latency. However, most importantly and differently from other schemes, hash-routing techniques have a set of key advantages. First, they are extremely robust against rapid variation of traffic patterns and traffic spikes, whether regarding specific locations or contents. Second, they evenly distribute traffic across a network domain, *de facto* eliminating the possibility of hot spots. Third, they are easier to model and as a result provide predictable performance and make cache placement easier. In addition, their robustness to peaks makes them better candidates for emerging ISP-CDN cooperative caching schemes, given that it does not require adaptive changes and this also makes it easier to keep SLAs under control. Fourth and of key importance, they provide several knobs to fine-tune performance metrics depending on the target topology, deployment models and expected workloads.

Chapter 5

On the design and implementation of high-speed caching nodes

5.1 Introduction

After discussing modelling and design aspects of distributed caching, focusing on how a set of caching nodes can interact to optimise performance, we now investigate how to improve the design and implementation of a specific caching node of a networked system.

In particular, we address the problem of how to optimise the design of read-through caching nodes, which are a fundamental building block of key networked systems such as CDNs and future ICN architectures such as CCN [89] and NDN [183]. A read-through caching node receives queries for specific content items in the form, for example, of an HTTP GET request or a CCN/NDN Interest. If it currently stores the requested content, it serves it to the requester from its caching space, otherwise it fetches the requested content from the origin or an upstream cache, stores it and finally serves it to the requester.

We aim to improve the state of the art by designing and implementing a cost-effective caching node capable of sustaining high cache hit ratio and line-speed operations. Our approach relies on using inexpensive commodity hardware together with highly optimised software. However, achieving these objectives is challenging. In fact, reactive read-through caches, differently from proactive content caches, store content items as they serve them. Therefore, they need to support concurrent reads and writes efficiently and make intelligent caching decisions to maximise cache hit ratio while being able to sustain line speed operations.

Arguably, the most important performance metric for a read-through cache is the throughput of traffic served from its cache memory, *i.e.,* the throughput originated by cache hits. In this respect, there are two key hardware properties limiting the extent of software optimisations. The first is the capacity of the cache memory as it directly impacts the number of distinct content objects that can be cached and therefore the cache hit ratio. The second is the external data rate or bandwidth of the memory, since from that depends the maximum read throughput that the cache memory can attain. However, building a system with large and fast memory using a single technology is very expensive. We address this problem by exploiting heterogeneous technologies, specifically DRAM and Flash-based SSD drives, which operate at different points of the cost/performance tradeoff. We devise novel techniques efficiently taking advantage

of specific characteristic of the heterogeneous memory available to improve performance.

Large amount of work already focused on designing key-value stores using a combination of DRAM and Flash memories (see for example [5], [7], [48], [50], [119]). However none of these designs are suitable for our purposes. In fact, the vast majority of them only support a store semantics and cannot be efficiently used for caching. Even the few designs that do support caching (*e.g.,* [5], [49]) are not optimised for high-speed caching and primarily target applications requiring very large datasets, such as data deduplication. As such, they focus on hardware configurations that minimise the cost/capacity ratio of the memory used and all assume a case in which DRAM is very small in comparison to Flash memory, to the point of being barely sufficient to store the index of contents stored in Flash. As a result, all these designs use the DRAM only to store the index and as a write buffer for the SSD, while all content items are stored exclusively in SSD. A substantial part of that work focuses on devising novel ways to compress the size of the index or to store part of the index on SSD while keeping read and write amplification under control.

Our key intuition is that in high-speed caching applications, a hardware configuration that primarily maximises the number of stored items is not always beneficial if the cache is not fast as well. In fact, a large memory can store many items and therefore lead to a higher cache hit ratio. However, if the memory cannot sustain the throughput required to serve all hits, the large availability of space is underutilised. On the other hand, a small but fast memory might not be able to operate at its full bandwidth, because its limited space does not allow to store enough contents to generate high cache hit ratio.

Table 5.1 shows the average three-year Total Cost of Ownership (TCO) of DRAM and SSD technologies normalised by capacity and bandwidth as of 2015. As it can be observed, *while it is more cost-effective to use SSDs to achieve a large capacity, DRAM is considerably more cost-effective to achieve high throughput*. Fortunately, the power-law distribution of content popularity, typical of content delivery workloads [25], makes it possible and very effective to use a combination of small and fast memory like DRAM to store a few very popular contents and a slow and large memory like SSD to store a larger number of less popular contents.

Table 5.1: Three-year TCO of DRAM and SSD memory technologies, including cost of hardware and energy at 0.10 USD/kWh

Memory	**Cost/capacity (USD/GB)**	**Cost/bandwidth (USD/Gbps)**
DRAM	6.72	0.63
SSD	1.1	56.2 [1]

Based on this tradeoff we design a system that uses both DRAM and SSD to store content in a way to efficiently exploit key properties of both. Differently from previous work we actively use DRAM to store hot content items (in addition to the index data structures) to take advantage of its low cost/bandwidth. We disregard HDDs in our design because of their very low throughput performance and high random access latency, which make them unsuitable for our purpose. In addition, SSD prices are dropping much faster

[1] Assuming large reads

than HDDs and, according to market forecasts [60], the launch of 3D NAND SSD drives scheduled for Q3 2016 will drive the cost of SSDs further down, to the point that both cost/capacity and cost/bandwidth of SSDs will be lower than HDDs by the end of 2016.

This chapter provides two main contributions. First, we examine a number of design options and derive general principles applicable to the design and implementation of caches for content distribution purposes, which could be adopted both in backend key-value caching systems (*i.e.,* `memcached` [130]), Content-Delivery Networks and packet-level caches envisaged by ICN proposals. Second, based on these general principles, we design and implement H2C, a hierarchical two-layer line-speed packet cache that we use as content store for CCN/NDN routers and evaluate its performance with trace-driven experiments.

We focus on the ICN use case because of its highly demanding technical challenges and also because it is the application that has received the least attention from the research community so far. Its technical challenges mainly stem from caching at packet-level granularity which requires to store a very large number of small objects and support a frequency of read operations considerably higher than object-level caches of CDNs. It should however be noted that most of the findings of this work are applicable to other use cases as well.

The remainder of this chapter is organised as follows. In Sec. 5.2 we provide general design guidelines for the design of two-layer DRAM/SSD caches. Based on these findings, we describe the design and implementation of H2C in Sec. 5.3. In Sec. 5.4 we evaluate the performance of H2C integrated with a CCN/NDN content router using trace-driven experiments. In Sec. 5.5 we survey related work and highlight the unique contributions of this chapter. Finally, we summarise our findings and present our conclusions in Sec. 5.6.

5.2 Design principles for two-layer DRAM-SSD caches

In this section, we identify a set of general principles for designing two-layer caches (DRAM-SSD) for content distribution. We explore various tradeoffs and validate our findings with trace-driven simulations.

5.2.1 Exploiting SSD characteristics

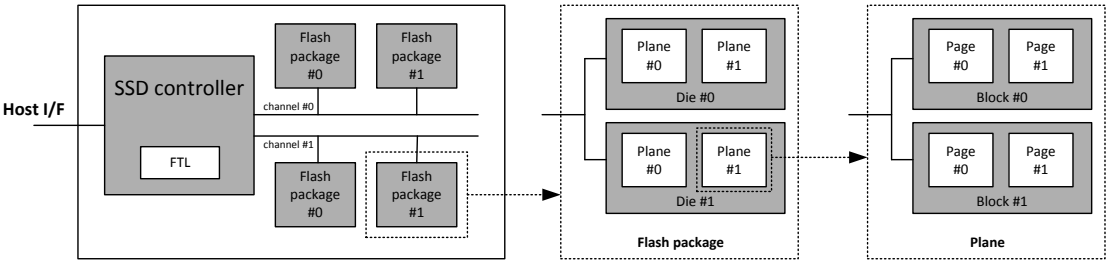


Figure 5.1: Internal architecture of an SSD drive

In order to design efficient caching systems using SSD, it is necessary to fully understand the key operational properties of SSD drives and how design decisions impact throughput and latency of read/write operations, as well as SSD durability. With this objective in mind, we overview the main SSD characteristics and derive a series of design guidelines for maximising performance of SSD drives for

caching applications.

Fig. 5.1 shows the internal architecture of an SSD drive. Hosts issue read/write requests for logical blocks to the SSD controller over a standard interface like SATA, SAS or PCIe. The SSD controller comprises a Flash Translation Layer (FTL) that translates logical blocks into physical pages. This mapping is executed according to sophisticated algorithms, generally not disclosed by manufacturers, which aim to improve throughput, latency and durability of the drive.

The SSD controller is connected to flash packages through a number of hardware channels. Each channel operates independently, so requests using different channels are processed in parallel, while requests using the same channel are serialised. Operations targeting different packages can be processed by each package in parallel. Each package contains several dies and each die can process requests independently to other dies in the same package. Each die comprises a number of planes. Planes belonging to the same die can perform operations in parallel but only as long as they are of the same type (read/write/erase). Each plane is then composed of several blocks, each consisting of several pages. Pages have normally a size of 2-16 KB and blocks contain 32-256 pages (*i.e.,* 64 KB - 4 MB). As the reader may have already concluded from this brief description, SSD drives have a high degree of internal parallelism that can be harnessed to optimise performance.

SSD latency does not generally depend on whether operations are executed sequentially or randomly because there is no performance penalty for a seek operation as in HDD technology. However, throughput does depend on sequential locality. In fact, FTLs normally map consecutive logical block addresses to physical pages belonging to separate channels and packages to fully exploit channel-level and package-level parallelism [37]. This is similar in principle to the striping mechanism used in RAID 0 systems.

A page is the minimum unit at which a read or write operation can be performed. Therefore executing a read or write for a subset of the data stored in one page requires reading/writing the entire page and therefore requires the same amount of time of executing the same operation on the entire page.

In addition, write operations can only be executed on empty pages, so, if a page needs to be overwritten, it must be erased first. However, erase operations can only occur at a block granularity. Therefore, overwriting a single page in a block requires first to erase the entire block and then to write the updated pages along with all other pages of the block in use at the time of the erase.

An erase operation incurs high latency. As a reference, on a standard Single-Level Cell (SLC) NAND flash, an erase operation normally requires 700 μs, as opposed to 250 μs of a write and 25 μs of a read [36]. Another issue with writes is that each SSD block can only undergo a limited amount of writes and erases (sometimes referred to as Program/Erase or P/E cycle) over its lifetime, normally up to a few 100K. For these reasons, Flash optimised file systems implement wear levelling mechanisms to evenly distribute writes over logical blocks. In addition, standard FTL controllers also implement measures to reduce the number of erase operations. For example, if a specific logical block page needs to be overwritten, a TFL can remap the logical block to an empty page, execute the write and then mark the old page as stale for a garbage collector process running in the background. As a result, overwrites can be executed without incurring the latency of a block erase. This technique can be applied even if the drive is full

of data. In fact, SSD drives are normally over-provisioned, *i.e.,* have greater capacity than advertised. The additional capacity cannot be used directly by a host, but it can be used by the SSD controller for improving performance and also to replace blocks damaged by too many P/E cycles.

After this discussion on SSD characteristics we can now derive a set of useful guidelines for efficiently using an SSD drive in a caching application.

Minimise writes to SSD. Any entity of the SSD (package, plane, block or page) cannot simultaneously process write and read operations. It follows that the amount of writes impacts the throughput available for reads. Since in caching, as opposed to storage, it is not necessary to guarantee the availability of an item, writes may be dropped. However, suppressing writes needs to be done carefully, to avoid compromising cache hit ratio. We discuss this issue more in detail in Sec. 5.2.2

Issue reads and writes no smaller than page size and aligned with SSD page boundaries. Since read and write operations cannot be executed with a smaller granularity than a page, to optimise throughput and latency it is necessary to group all reads and writes to fit in exact multiples of a page and to be aligned to page boundaries. Most importantly, even in the case of overwriting, executing requests aligned to page boundaries, but not aligned to block boundaries, does not affect performance because, as explained above, FTLs can remap logical blocks to perform writes in free pages. This may however increase wear, depending on the implementation of the FTL.

Issue many parallel reads and writes. Issuing many read and write requests in parallel helps to fully exploit the high degree of internal parallelism of SSD drives. It also helps to saturate the communication pipeline by keeping many operations in flight.

5.2.2 Selective cache insertion

As discussed above, the cumulative read/write throughput of SSDs is limited. Therefore, to maximise SSD read throughput it is necessary to minimise the amount of writes operations performed. However, such a write reduction needs to be achieved without compromising SSD cache hit ratio, since reducing the frequency of writes to SSD may adversely impact the freshness of stored contents and hence affect hit ratio. Therefore, designing a strategy to reduce SSD writes would need to take into consideration target traffic characteristics.

Typical Web content retrieval workloads are characterised by a high fraction of contents which are requested only once over a long period of time (sometimes referred to as *one-timers* or *one-time wonders*). As an example, Akamai reported that in the traffic served by their infrastructure, one-timers account for 74% of requested objects [122]. Also EdgeCast reported similar patterns, although in their case the fraction of one-timers corresponds to 60% of requested contents [152]. We identified similar patterns also in traffic traces at our disposal. Analysing a daily trace of URL requests to Wikipedia [171], we observed that 69% of content items were requested only once.

The high frequency of one-timers could be exploited to improve cache hit ratio and reduce SSD writes. Akamai use a cache insertion policy named *cache-on-second-hit* to prevent caching one-timers [122]. According to this technique, a content item is cached only the second time it is requested, hence

eliminating the possibility of a one-timer being inserted. This technique is implemented using a Bloom filter stored in DRAM. Each time a caching node receives a request, the Bloom filter is looked up for the URL of requested content. If the content URL is in the Bloom filter, the content is cached. Otherwise the URL is added to the filter and the content is discarded. Applying this technique to Akamai operational infrastructure improved cache hit ratio from 74% to 83% [122].

The Bloom filter implementation of this technique requires a modest memory overhead (66.8 MB in Akamai's case). However, a Bloom filter lookup/insertion at each content request causes a non-negligible computational overhead. In fact, that requires few hash function computations and load/store operations in random areas of the Bloom filters. Since Bloom filters of that size are too large to fit entirely in typical caches of current commodity CPU (*e.g.,* L3 cache of high-end Intel Haswell processors is 20 MB) and read/writes take place in random positions with no correlation between subsequent lookups, such operations result in very frequent CPU cache misses and consequent higher-latency DRAM memory accesses. All this causes low CPU utilisation as a consequence of frequent stalls waiting for load/store operations. For this reason, this implementation may not be suitable for line-speed operations with small content sizes, which is the case, for example, of CCN/NDN routers.

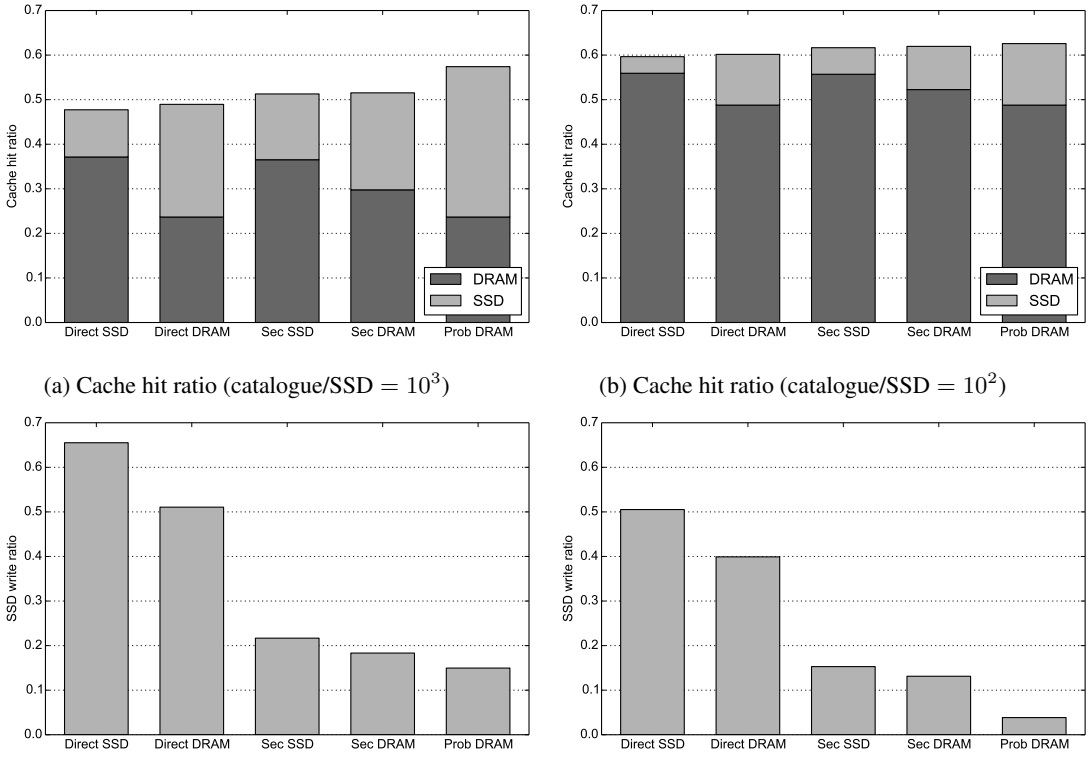
To address this limitation, we design a novel lightweight cache insertion policy which we named *Probationary insertion*, which does not incur the memory and computational overhead of the *cache-on-second-hit* technique. According to this policy, whenever an item is requested which is not currently in cache, it is stored in DRAM independently of how many times it was requested before. Both DRAM and SSD caching space are managed using independent LRU replacement policies. Each item in DRAM is associated to a flag bit which is initially unset when inserted. If the content item is requested while in DRAM, the flag is set and the content is moved to the top of the LRU list. When the content reaches the bottom of the LRU list, if the flag is set, the item is moved to SSD and inserted at the top of the SSD LRU list, otherwise it is discarded. When a content stored in SSD is requested, it is copied to DRAM and inserted at the top of the DRAM LRU list. In practice, a content is inserted in SSD only if, during its sojourn in DRAM, it is requested at least once after its insertion. This policy can be managed using for example a hash table like the one used in Caesar [138], which requires at most one hash computation and one memory access per request. In addition, using prefetching mechanisms like the one used by CuckooSwitch [185], the frequency of DRAM reads can be considerably reduced.

We evaluate the performance of the proposed probationary insertion policy and compare it against two implementations of Akamai's cache-on-second-hit rule, which we named, respectively *Second-hit SSD* and *Second-hit DRAM*. In both implementations DRAM and SSD caching space are managed according to the LRU replacement policy. When a content stored in SSD is requested, it is copied to DRAM and placed at the top of the DRAM LRU list. When a content stored in DRAM reaches the bottom of the DRAM LRU list, it is demoted to SSD. The only difference between the two implementations is that in Second-hit SSD, a content is inserted (when requested for the second time) in SSD, while in Second-hit DRAM, the content is inserted in DRAM. The second-hit SSD implementation ensures that contents stored in DRAM have a greater probability of being hit. In fact, a content, in order to be placed

in DRAM, needs to be requested at least three times (two to be inserted in SSD and one to be promoted to DRAM). However, by inserting contents in SSD without a previous sojourn in DRAM may lead to a greater number of SSD writes. The second-hit DRAM implementation has opposite advantages and disadvantages.

To better understand the impact of these three selective cache insertion policies, we also evaluate their performance against two baseline policies not performing any filtering, which we named *Direct SSD* and *Direct DRAM*. They are similar to the second-hit policies described above, with the only difference that contents are always inserted.

We evaluate the performance of these five schemes with trace-driven simulation using the Wikipedia workload used in Chapter 4. We assume that the ratio between SSD and DRAM size is 10 and we evaluate the cases in which the ratio between content catalogue and SSD size equals to 10^2 and 10^3. We measure both cache hit ratio (measured from both DRAM and SSD) as well as SSD write ratio, which corresponds to the ratio of content requests causing an SSD write operation.



(a) Cache hit ratio (catalogue/SSD $= 10^3$)

(b) Cache hit ratio (catalogue/SSD $= 10^2$)

(c) SSD write ratio (catalogue/SSD $= 10^3$)

(d) SSD write ratio (catalogue/SSD $= 10^2$)

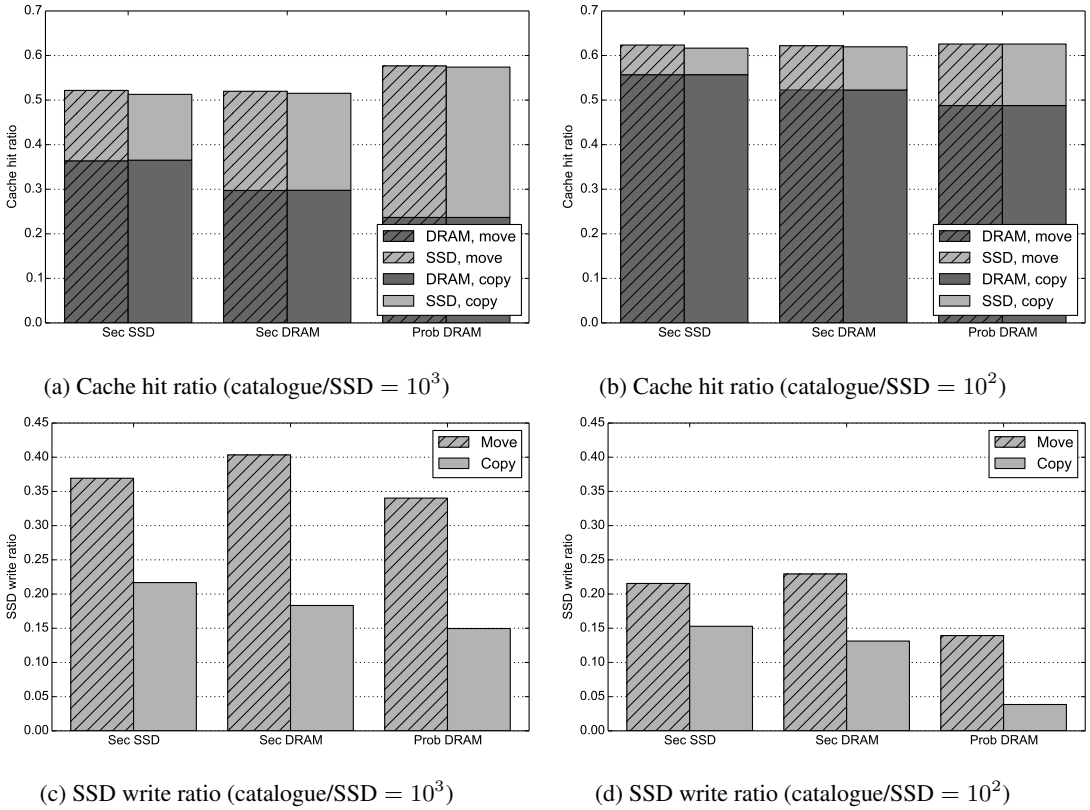
Figure 5.2: Performance of selective cache insertion algorithms

The results, depicted in Fig. 5.2 validate the soundness of our design. First of all, the proposed probationary DRAM insertion policy outperforms all other policies in terms of both cache hit ratio and SSD write ratio. All three selective insertion policies outperform the two indiscriminate insertion ones in terms of both cache hit ratio and SSD writes under all scenarios investigated. While cache hit ratio improvements are marginal, reductions in SSD writes are substantial. This is very important since SSD write reduction impact overall system costs, since it would allow to achieve a target read throughput using

fewer SSD drives. Finally and expectedly, Direct and Second-hit SSD policies yield a greater DRAM hit ratio than their DRAM insertion counterparts, although they suffer of more frequent SSD writes.

The main results are that selective cache insertion policies are very effective in substantially reducing SSD writes without impacting cache hit ratio, actually slightly improving it. Among all policies considered, our proposed probationary DRAM insertion yields better performance in terms of both cache hit ratio and SSD writes and also incurs less memory and computational overhead than insert-on-second-hit implementations. For all these reasons, we adopt probationary DRAM insertion in the design of H2C.

5.2.3 Copy to DRAM on SSD hits



(a) Cache hit ratio (catalogue/SSD $= 10^3$)

(b) Cache hit ratio (catalogue/SSD $= 10^2$)

(c) SSD write ratio (catalogue/SSD $= 10^3$)

(d) SSD write ratio (catalogue/SSD $= 10^2$)

Figure 5.3: Performance of copy and move operations

Given the greater throughput and lower latency offered by DRAM in comparison to SSD, it is desirable that most popular content items are placed in DRAM. A common approach to achieve this objective is to promote contents present in SSD to DRAM when requested.

Upon promoting a content to DRAM, the copy of the content in SSD can be either removed to make room for new content or maintained. Leaving a copy on SSD has the advantage that, if the content moved to DRAM is no longer requested and needs to be demoted to SSD, a write to the SSD is not required since a copy of the content is already there. However, this approach has the disadvantage of reducing the number of distinct content items that a node can cache, since part of the SSD memory is used to store contents already available in DRAM. This may degrade cache hit ratio performance.

In this section, we examine these two design options and evaluate how they impact cache hit ratio

and frequency of SSD writes. We perform our evaluation using the same setup used in Sec. 5.2.2 and evaluate how copying or moving an item to DRAM upon SSD hits impact performance under Second-hit SSD, Second-hit DRAM and Probationary DRAM insertion policies.

The results, depicted in Fig. 5.3 show that leaving a copy in SSD after hit causes a negligible decrease in SSD cache hit ratio under all scenarios considered. However, it widely decreases the amount of SSD writes required. For this reason, in the design of H2C, we opted for copying items to DRAM when an SSD hit occurs rather than moving them.

5.3 H2C design

In this section, we build on the generic principles presented above to design and implement H2C, a hierarchical two-layer packet cache for CCN/NDN content routers that is able to serve traffic at a line speed of 20 Gbps.

We start by describing its architecture (Sec. 5.3.1) and then move to describe the optimisation techniques adopted (Sec. 5.3.2), the design of data structures (Sec. 5.3.3) and finally the packet processing flow (Sec. 5.3.4).

5.3.1 Architecture

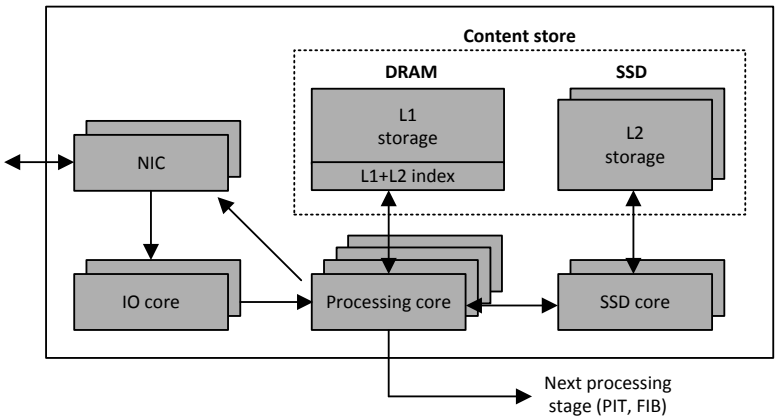


Figure 5.4: H2C architecture

H2C is a multithreaded userspace application designed for *nix operating systems running on a x86 architecture. An H2C instance runs on as many threads as the number of CPU cores available in the system. Each thread is bound to a specific core using the core affinity functionalities offered by the OS.

Cores are grouped into three sets depending on the task they perform (see Fig. 5.4).

Network I/O cores. They are in charge of fetching received packets from NICs using poll-mode drivers and forward them to a specific processing core based on the hash of the content identifier.

In H2C content objects are allocated to processing cores based on the hash of the content identifier to ensure that a content is always processed by the same core. This effectively shards contents among cores. As a result, each core can maintain dedicated data structures and process packets independently of other cores.

It should be noted that the role of Network I/O cores may be fulfilled in certain situations by NICs. Modern NICs support Receiver Side Scaling (RSS) functionality that places received packets on specific hardware queues depending on the result of a hash function computed on a number of fields of the received packets. However most commodity NICs limit the choice of fields that can be used to compute the hash. For example, Intel 82599 NICs, which we used in our experiments, only support hashing based on IP v4/6 source/destination addresses and TCP/UDP source/destination ports. This limitation could be overcome with the cooperation of other network entities. For example, CCN/NDN packets could be encapsulated in UDP packets. Network entities could agree on a range of UDP destination ports to use and select a specific port in such range based on the hash of the content item.

Packets are passed from Network I/O to processing cores using concurrent lock-less ring queues. These queues are implemented using the atomic compare-and-swap (CAS) instruction which is widely supported by modern CPUs, including x86 architectures. This enables lightweight producer/consumer synchronisation without recurring to locks. It should be noted that packets are not copied during this operation. Cores only pass each others 32-bit pointers indicating the index of a packet in a packet buffer pool.

Multiple instances of Network I/O cores do not need synchronisation among each other, since each core has exclusive access to a dedicated hardware queue on each NIC.

Processing cores. They are in charge of receiving packets from NICs or Network I/O cores and performing cache lookup operations. As mentioned above, since we shard packets across cores, each processing core can operate using dedicated instances of all data structures. Also, each processing core has exclusive access to an area of the SSD memory for storing data. In case NIC's RSS functionality is used (instead of software-based hashing by Network I/O cores), each processing core also has exclusive access to a DRAM region, otherwise DRAM is shared by all cores. However, even in the latter case, synchronisation overhead is minimum as allocation/release of packets from the packet pool is also implemented using the atomic CAS instruction.

When an Interest for a chunk is received, the processing core looks it up in the content cache. If available in the DRAM cache, it sends it to the destination NIC directly using a DMA transfer. If available in SSD cache, it issues a request to an SSD I/O core to fetch the content from SSD. If the content is not in cache, then it performs other standard processing operations (*i.e.,* PIT and FIB lookups).

SSD I/O cores. Their only functionality is to receive SSD read/write commands from processing cores and execute them. All communications between processing and SSD I/O cores occur using the same kind of lock-less rings used for communications between NIC I/O and processing cores. We use dedicated cores to access SSDs in order to separate high-latency SSD lookup operations from forwarding and DRAM lookup operations in order not to compromise the latency of the latter.

One key design challenge is how to deal with packet-level caching (each addressable content packet

is referred to as a *chunk* in CCN/NDN terminology). In fact, in addition to requiring considerably more lookup operations than object-level caching, reading and writing chunks randomly to SSD causes poor performance for two reasons. First, chunks may be smaller than SSD pages and therefore lead to read and write amplification if stored independently. As a reference, NDN specifications mandate chunk sizes of up to 4 KB, while modern SSD drives may have pages of up to 32 KB. Second, even if page and chunk size matched, performing small random read and write operations does not fully exploit the parallelism offered by SSD drives.

However, since it is very likely that a user requesting a content object does so by requesting all chunks sequentially from first to last, we can use this pattern to optimise performance. We design data structures to process groups of consecutive chunks together. We name this group of consecutive chunks a *segment*. When H2C reads/writes data from/to SSD, it does so at a segment granularity. H2C uses SSD drives as raw disks, *i.e.,* it accesses them reading/writing at logical block addresses bypassing the filesystem. Using a standard feature of *nix kernels named vector I/O or scatter/gather I/O, it is possible to copy chunks belonging to the same segment between random locations on DRAM and a contiguous location in SSD. As a by-product, organising chunks in segments reduces the memory overhead of control data structures, as we will see in Sec. 5.3.3. The size of the segment depends on various parameters, but it should be at least as large as the size of an SSD page. In our experiments we use chunks of 4 KB and segments of 8 chunks, *i.e.,* 32 KB.

As the reader may have already noted, a key design decision is the extensive use of sharding throughout the architecture, specifically to partition DRAM and SSD regions and assign each of them exclusively to one processing core. This decision is motivated by the findings of Chapter 3, which showed how a system of hash-partitioned caches yields the same cache hit ratio performance of a single cache of cumulative size. Not only sharding can be safely applied without compromising cache hit ratio, but it also simplifies system implementation and improves throughput since processing cores can operate independently without requiring synchronisation to access shared memory regions. In addition, the decision of assigning chunks to shards based on segment identifier, as opposed to content object identifier, has been made to reduce load imbalance, in accordance to the results of Theorem 3.4.

5.3.2 Optimisation strategies

To achieve the objective of line speed operation, we made a number of design decisions enabling to optimise performance by (i) addressing inefficiencies of *nix operating systems and (ii) exploiting intrinsic characteristics of SSD drives.

5.3.2.1 Exploit parallelism

In order to maximise throughput, we design H2C with the objective of exploiting the high degree of parallelism offered by modern commodity hardware at all layers of the system. First, we use multiple hardware queues made available by modern NICs to allow multiple CPU cores to send/receive packets to/from NICs in parallel without requiring any synchronisation among cores. Multiple CPU cores process packets in parallel independently. Since modern CPU have separate execution units for arithmetic and memory load/store operations, we designed our packet processing algorithms to interleave as much as

possible computation tasks (*i.e.,* hash computations) and memory access tasks, to exploit internal CPU parallelism. We also allocate packet buffers in DRAM aligned in such a way that consecutive packets can be fetched using different DRAM channel, hence improving DRAM throughput. We also exploit parallelism provided by modern Non-Uniform Memory Access (NUMA) architectures, by placing all data in the local NUMA node accessed by each core, to prevent remote NUMA memory accesses. Also, we read/write data to SSDs in batches to exploit internal parallelism of SSDs and use multiple SSDs in parallel to improve throughput.

5.3.2.2 Zero-copy operations

In *nix kernels, each I/O operation between a userspace memory buffer and an I/O device requires an intermediate memory copy to a kernel buffer. Despite modern DRAM has very high bandwidth, executing memory copies for all network traffic handled at several tens of Gbps would be detrimental to system performance. We therefore design our system to support zero-copy operations throughout the entire packet processing pipeline. To avoid memory copies in NIC I/O operations, we use poll-mode NIC drivers that support DMA transfer of data between NIC and a userspace memory buffer, hence bypassing the OS kernel. We also perform zero-copy operations in accessing the SSD by using the Direct I/O feature of *nix kernels. A consequence of using Direct I/O for SSD access is the unavailability of OS page cache, scheduling and read-ahead mechanisms offered by *nix kernels. However, this is not a problem since H2C already takes care of all these performance optimisation aspects natively and more effectively than the kernel, given its better understanding of the workload.

5.3.2.3 Batch processing

We design our system to process all I/O tasks (towards both NICs and SSDs) in batches in order to reduce per-packet processing overhead. This has two desirable effects. First, it reduces processing overhead by amortising the cost of expensive system calls over multiple packets. Second, it leads to a better I/O performance by allowing NICs and SSDs to group multiple packets into a single PCIe transaction and also have a greater number of data in flight at any given time, improving throughput. Both NIC-CPU and SSD-CPU communication pipelines cannot be effectively saturated by transferring one packet at a time, resulting in limited throughput.

A side effect of batched operations is the latency increase, especially at low load. We mitigate this problem by setting timeouts to cap the maximum waiting time.

5.3.2.4 Aggressive prefetching

We perform aggressive prefetching of data at all layers of the system to mask memory access latency of both DRAM and SSD.

DRAM prefetching is made possible by processing packets in batches. When a processing core fetches a batch of packets from Network I/O cores (or directly from NICs), it prefetches a window of packet headers from DRAM to CPU L1 cache before processing them. Subsequently, it alternates processing a prefetched packet and prefetching a new one until the entire batch is processed. By properly dimensioning this prefetching window, it can ensured that a packet header is in L1 cache by the time it

needs to be processed. However, the size of the prefetching window is a delicate parameter to set. An excessively small window may result in data being prefetched too late to be available in L1 cache in time for processing, resulting in CPU cache misses. On the other hand, an excessively large prefetch window would thrash the CPU L1 cache, which has a limited size (32-64 KB on modern x86 architectures), eventually leading to cache misses and decreased performance.

SSD I/O cores also prefetch data from SSD to DRAM to mask SSD access latency. When a request arrives for a chunk currently in SSD, the chunk is copied to DRAM along with a number of subsequent chunks of the content object, so that requests for subsequent chunks would result in a DRAM cache hit, hence resulting in lower latency. This of course assumes that chunks of a content object are requested by users sequentially.

5.3.2.5 Huge pages

The CPU needs to maintain a page table mapping virtual to physical memory addresses. To accelerate physical address resolution, the CPU caches page mapping entries in a Translation Lookaside Buffer (TLB). TLB misses cause a latency penalty in accessing a memory location.

The standard size of a memory page on modern CPUs is 4 KB. Since the TLB has a limited number of entries, in systems with a large amount of memory a substantial amount of TLB misses may occur. To overcome this issue, we use a feature supported by modern CPUs and modern OSes named *Huge Pages* that supports using 2 MB pages, which, in comparison to 4 KB pages reduce the number of TLB entries by a factor of 512. This considerably reduces the rate of TLB misses and improves memory access performance.

5.3.2.6 Sharding

In order to avoid lock contention, we partition the memory space of both DRAM and SSD in as many regions as the number of processing cores and assign each region to a specific core. The Network I/O cores or, if available, the NIC RSS mechanisms ensure that Interest and Data packets for a specific chunk are always handled by the same core, by mapping chunks with a hash function computed on segment identifiers (by using a concatenation of object name and integer part of chunk_id/segment_size). As a result, read and write operations involving a particular chunk are always performed by the same core, therefore eliminating the need for locks on most operations. This also improves the performance of CPU cache because it reduces cache invalidations. In turn, such memory partitioning technique also enables NUMA-aware memory allocation, reducing DRAM access latency. In the light of the considerations made in Sec. 3.3.4, H2C shards items among processing cores on a segment granularity as opposed to a content object granularity to improve load balance.

5.3.2.7 Optimised SSD access

To access the SSD drives, we use a combination of Direct I/O, Vectored I/O and Asynchronous I/O, which are all standard *nix I/O techniques enabling zero-copy batched transfers between DRAM and SSD. Direct I/O enables direct transfer of data between a userspace memory buffer and SSD without an intermediate copy into a kernel buffer, hence enabling zero-copy operations. Vectored I/O, sometimes

referred to as Scatter/Gather I/O, makes it possible to copy data scattered across various areas of a userspace memory buffer into a contiguous area in SSD and viceversa. This is done with a single system call and with virtually no performance penalty compared to fully sequential operations (*i.e.,* between a contiguous memory region in DRAM and a contiguous memory region in SSD). We use this technique to transfer chunks belonging to the same segment together to avoid read/writes smaller than a page and, in the case of reads as a way to prefetch chunks very likely to be requested at close distance. Finally we use Asynchronous I/O to batch read/write requests for various segments in a single system call. Not only this helps amortise the cost of a system call over multiple segments, but also helps filling channels and therefore improving attainable throughput.

This SSD interfacing mechanism is similar in principle to the one proposed in [154]. This is however just a building block of our complete end-to-end system.

5.3.3 Data structures

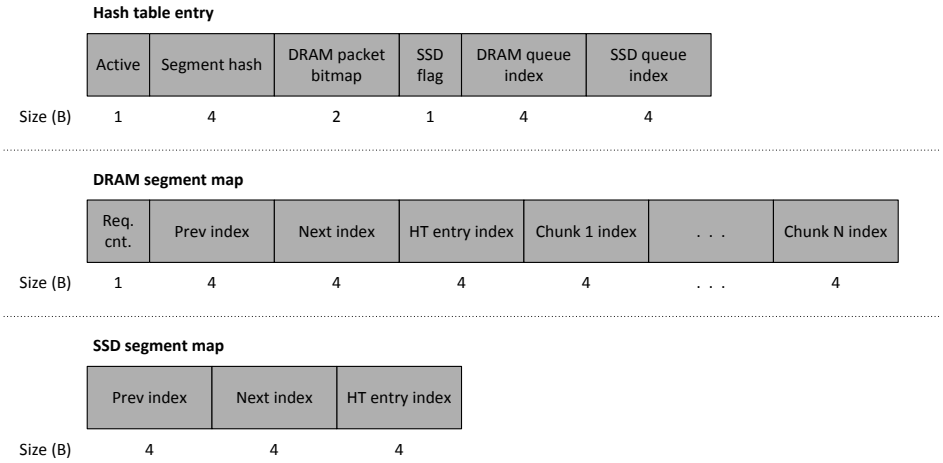


Figure 5.5: H2C data structures

H2C uses a number of data structures to manage the two-layer cache (see Fig. 5.5). Each processing core has a dedicated instance of each data structure. This means that no data structures are concurrently accessed by cores, with the exception, of course, of the lockless rings used to pass information between Network I/O cores and processing cores and between processing cores and SSD I/O cores.

The main data structure is a *hash table* used for indexing all the segments stored in the portion of the cache (DRAM+SSD) assigned to the corresponding core. It is implemented using an open-addressed hash table similar to the one used in [138]. The goal of this hash table design is to minimise the access latency. Therefore, it has been designed to enable the execution of a lookup/insertion with at most a single DRAM memory access. This was achieved by dimensioning buckets to a cache line (*i.e.,* 64 B in our architecture) and storing up to four entries in a single bucket. In such a way, even if the bucket is not in CPU cache, it can be retrieved with a single memory read which copies the entire bucket CPU L1 cache. At this stage, even in case of collisions (*i.e.,* multiple entries in the same bucket), it is possible to locate the desired entry just by iterating the bucket, which is in L1 cache and can be accessed very quickly. It may still occur to have more than four colliding entries, in which case they cannot all fit in a bucket. This

case can be managed using chaining, but it is expected to be rare if the hash table is well dimensioned.

Each entry of the hash table is composed of 6 fields for a total of 16 B and comprises the following fields.

- An *Active* field, indicating whether the entry is valid or not. This entry is used for fast deletion (*i.e.,* to remove an entry from the hash table we set this byte to 0 rather than deleting the whole entry).
- A *segment hash* field, which is a 32-bit hash value of the segment identifier.
- An *SSD flag*, indicating whether the entire segment is stored in the SSD or not.
- A *DRAM packet bitmap*, indicating which packets of the segment are stored in the DRAM.
- Finally, a *DRAM segment map pointer* and a *SSD segment map pointer*, which store a pointer to the associated DRAM/SSD segment map respectively.

Every hash table entry points to a *DRAM segment map* and/or a *SSD segment map* depending on where the packet is cached.

The *DRAM segment map* contains primarily an array of N elements (*i.e.,* the number of packet in a segment) pointing to the buffers storing the packets belonging to a given segment. DRAM segment maps of various segments are organised as a linked list and ordered according to the LRU policy. We call this list *DRAM replacement queue*. Each DRAM segment map contains the following fields.

- A *Request counter* field, counting how many times, R, an item has been hit since insertion in DRAM. This is used to enable the probationary insertion mechanism described in Sec. 5.2.2
- *Previous* and *Next* fields, storing the indices of the DRAM segment maps precending and following this element in the LRU queue.
- A *Hash table pointer* pointing back to the hash table entry associated to this segment. This pointer is needed to make it possible to update the hash table entry upon eviction of a segment from DRAM.
- An array of pointers to all the buffers storing the chunks belonging to a segment.

The *SSD segment map* only contains pointers to previous and next elements as well as a pointer to the associated hash table entry. SSD segment maps of various segments are organised as a linked list and ordered according to the LRU policy. We call this list *SSD replacement queue*. It should be noted that while a DRAM segment map stores the address of each chunk, the SSD map does not store any pointer to the related segment address. This is because in DRAM chunks belonging to a segment are not necessarily stored in contiguous locations. In SSD all chunks of a segment are stored contiguously and their position in SSD reflects the position of their associated SSD segment map in the SSD replacement queue. Therefore the SSD segment map pointer stored in the hash table entry is sufficient to locate the segment in SSD.

To manage free segments in SSD, each processing core uses a FIFO queue of free segments in the SSD, which we named *SSD segment pool*. When a processing core needs to request the insertion of a

segment to SSD to an SSD I/O core, it inserts it at the location indicated at the head of the queue. When a segment is removed from SSD, its location is appended to the tail of the queue. It is important to note that to avoid concurrent access to the main hash table between SSD I/O and processing cores, all state about SSD is maintained by processing cores, which provide SSD I/O cores with all memory addresses required to perform the requested I/O operations.

Finally, all data packets in DRAM are stored in a lock-free pool of pre-allocated packet buffers named *DRAM packet pool*. Whenever a packet is retrieved from the NIC or the SSD, it is stored in DRAM using a free buffer of the packet pool. Conversely, when a packet is evicted from DRAM, the corresponding buffer is added back to the packet pool. It should be noted that the memory used by the DRAM packet pool is entirely allocated at startup time and the allocation and release of packet buffers is managed in userspace. This is done to avoid the overhead of per-packet calls to `malloc` and `free` system calls.

The design of the control data structures of H2C exploits the fact that chunks belonging to the same content are likely to be requested sequentially to reduce the frequency of DRAM memory access to perform an H2C lookup. Since looking up chunks belonging to the same segment requires access to the same hash table bucket and same DRAM/SSD segment map, when chunks are looked up sequentially and close in time to each other, the related memory entries will be fetched from DRAM only when the first chunk of the segment is looked up. When subsequent chunks of the segment are looked up, there is a high probability they will be found in CPU L1 cache.

In conclusion, we note that in the design of the data structures we focused primarily on reducing processing overhead given the line speed operations objective, sometimes at the cost of losing memory efficiency. However, despite this, the memory footprint of the data structures used to manage H2C is still modest, as we show in Tab. 5.2.

Table 5.2: Memory overhead of H2C control data structures, assuming chunk size of 4 KB and $N = 8$

	DRAM		
SSD	**16 GB**	**32 GB**	**64 GB**
128 GB	146 MB	169 MB	214 MB
256 GB	270 MB	293 MB	338 MB
512 GB	518 MB	541 MB	586 MB

5.3.4 Packet processing flow

In this section we describe the packet processing flow of H2C operating as content store for a CCN/NDN router. As already mentioned in Chapter 2, CCN/NDN uses two types of packets: *Interest* and *Data*. The first is a request that a client uses to fetch a specific packet-sized chunk of data. The second is the data itself that satisfies the corresponding request. Both packets contain the name of the chunk (*i.e.,* content object name plus packet identifier) they refer to.

In the following we detail the packet processing flow for both Interest and Data packets.

First, packets are extracted in batch from NIC's hardware queues by Network I/O cores. Each Network I/O core that receives the batch of Interest and Data packets dispatches them to different processing cores according to the hash value of their segment identifiers (*i.e.,* $\lfloor \text{chunk\_id}/N \rfloor$) where N is the number of chunks of a segment) so that packets belonging to the same segment are always assigned to the same processing core. As already mentioned, this task could be taken care by NICs directly if their RSS functionalities support that or if content object identifier information is encoded in transport layer destination port.

Processing cores receive batches of packets and process them according to their type.

If the received packet is an Interest, a lookup in the segment hash table is performed using the segment identifier hash value already computed. In case the element is not present in the segment hash table, then PIT and FIB are looked up. In case the element is present, the processing core checks whether the packet is stored in the first (DRAM) or in the second (SSD) level cache using the DRAM packet bitmap and the SSD flag. If the packet is stored in DRAM, the corresponding Data packet is prepared and sent back to the requester and the request counter field is increased. The segment entry is also moved to the top of the DRAM replacement queue. If the packet is stored in SSD, the request packet is passed to an SSD I/O core that fetches the entire segment to which the requested chunk belong from the SSD and stores it in free DRAM packet buffers. Once the segment is moved to the first level cache, the SSD I/O core notifies the processing core that creates a new DRAM map entry and prepares the Data packet for transmission. When a chunk is copied from SSD to DRAM, the associated DRAM and SSD segment map entries are moved to the top of the DRAM and SSD replacement queues respectively. Notice that, motivated by the findings of Sec. 5.2.3, segments copied from the second to the first level of the cache are not removed from the SSD.

If the received packet is a Data and it is not already in the first or second level cache of H2C, a segment hash table entry is created and the corresponding DRAM segment map entry is created and inserted at top of the DRAM replacement queue by the target processing core. The data is then sent to the requester.

During those packet processing operations, segment eviction from DRAM or SSD cache may be required to make space available for new chunks received from the network or copied from SSD. When the DRAM cache is full, the segment at the tail of DRAM replacement queue is evaluated. If the value of its associated request counter R is greater than or equal to a predefined threshold value R_{min}, the segment is demoted to the SSD cache and placed at the top of the SSD replacement queue, otherwise it is discarded. As discussed in Sec. 5.2.2, setting a threshold $R_{min} = 2$ (counting the initial insertion as a hit) is sufficient to achieve a considerable reduction in terms of SSD writes without affecting SSD cache hit ratio. However, as shown in Sec. 5.4.5, a greater threshold value may also be used to further increase read throughput although at the cost of reducing cache hit ratio. Also, it should be noted that a segment is discarded if it is not full, *i.e.,* if not all its chunks have been received. This ensures that space in SSD is well utilised and avoid read and write amplifications. As a side effect though, it penalises content objects smaller than a segment as they will never be inserted in SSD.

When the SSD cache is full, the element at the tail of the SSD replacement queue is removed from the list. It is also removed from the segmented hash table if it is not stored in DRAM replacement queue.

It should be noted that, to reduce processing overhead, an entry of the segment hash table is deleted simply by setting the Active field to 0. Similarly a DRAM segment map is deleted by removing it from the DRAM replacement queue, returning all packet buffers to the pool and setting the DRAM packet bitmap of the associated hash table entry to 0. Finally, an SSD segment map is deleted by removing it from the DRAM replacement queue, returning the segment to the SSD segment pool and setting the SSD flag of the associated hash table entry to 0.

5.4 Performance evaluation

After describing the design of H2C, we now present an evaluation of its performance using trace-driven experiments.

We first investigate SSD performance in isolation and then use the insights gathered to fine-tune SSD configuration parameters and evaluate overall H2C performance. It should be noted that as our focus is on H2C performance, we do not analyse PIT and FIB lookup operations (which our prototype is capable of) that are performed after a content store miss.

5.4.1 Implementation

We implemented a complete CCN/NDN content router comprising name-based FIB, PIT and using H2C as content store.

We implemented our router as a Linux userspace application for an x86 CPU architecture. The code comprises ∼4000 lines of C code. The implementation makes extensive use of the functions provided by the Intel DPDK library [86], which implements many functionalities required by our design (*e.g.,* Huge Pages support, zero-copy operations and poll-driven NIC drivers) as well as optimised implementations of a number of useful functions efficiently using x86 instruction set.

We implemented the FIB according to the design of Caesar [138] and the PIT according to the design of [29]. Despite both designs were proposed for a network processor architecture, they are quite flexible and could be easily ported to an x86 architecture. We did however need to re-dimension some data structures to account for a shorter cache line (128 B in the network processor design *vs.* 64 B in the x86 architecture).

We enabled Jumbo frame support and configured the content router to support content chunks up to 4 KB, in line with NDN specifications.

5.4.2 Setup and methodology

We deployed our content router with H2C as content store on a general purpose server running Linux Ubuntu 12.04 LTS and equipped with two 4-core CPUs (Intel Xeon E5540 2.53 GHz, Nehalem architecture), 32 GB of DRAM, one Intel 82599 dual-port 10GbE NIC connected via a PCIe interface, and two 200 GB SAS SSD drives. The machine comprises two NUMA nodes, each being allocated one CPU and 16 GB of DRAM.

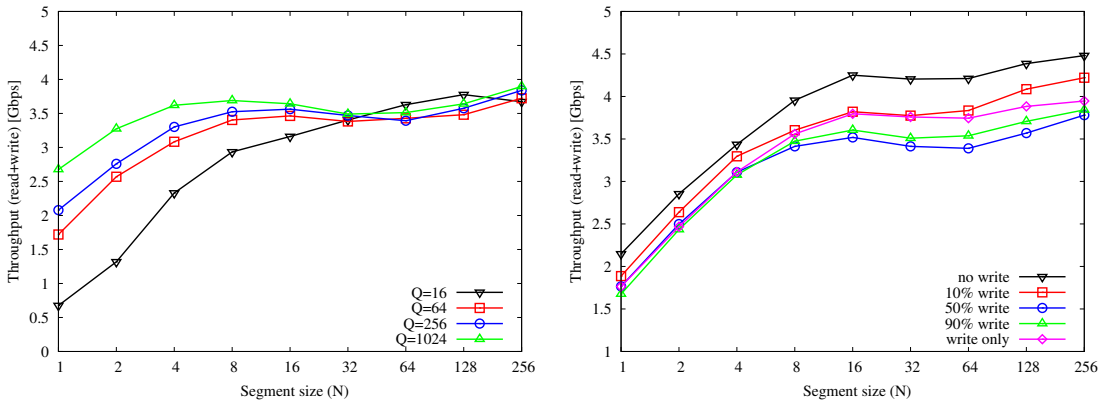
For our evaluation we connected H2C via direct optical fiber connection to another server with same

hardware configuration running a software traffic generator that we built specifically for this use case also using Intel DPDK.

As evaluation workload we used the Wikipedia trace we used in Chapter 4, which comprises ∼11.5 million requests over a catalogue of ∼1.8 million content items. We artificially vary the average content size from 1.5 MB to 15 MB to study the impact of different catalog sizes on H2C performance. The traffic generator issues requests for chunks of each content object sequentially, from first to last chunk and randomly interleaves requests for chunks of different objects.

Unless otherwise specified, H2C is configured with a DRAM cache of 20 GB, an SSD cache of 200 GB, and segments composed of $N = 8$ consecutive 4 KB packets (*i.e.,* segments are of 32 KB each). We enable the HyperThreading functionality (which results in 8 virtual cores per CPU being available to the OS). We allocate 14 virtual cores for packet processing and the remaining 2 virtual cores for SSD I/O tasks. We encapsulate ICN packets into UDP datagrams with appropriately set destination port based on content hash. As a result, processing cores can retrieve chunks directly from NICs without the intermediation of any Network I/O core. It should be noted that using SSD drives with larger capacity would increase the caching efficiency but would not impact the insights presented in the following sections.

5.4.3 SSD performance tuning



(a) SSD throughput *vs.* Q and N

(b) SSD throughput *vs.* read/write mix and N

Figure 5.6: SSD throughput *vs.* segment size (N), queue size (Q) and read/write mix

Before evaluating the overall throughput achieved by the H2C implementation, we examine the impact of SSD I/O parameters on system performance. We analyse two key parameters: the number of chunks per segment (N) and the queue size (Q), which corresponds to the number of segment read/write requests issued per I/O call.

These two parameters can heavily impact system performance. In fact, while a large value of N and Q can very likely improve throughput performance by enabling larger sequential read/write operations and effectively saturating the SSD I/O pipeline, excessively large values may detriment other performance metrics. Increasing Q and N impacts latency as each chunk I/O operation will have to wait for a longer batch of segment to be formed before a request can be issued. Also, increasing segment size N would likely impact performance of workloads with many small content objects, as such objects could not be

stored in SSD.

In Fig. 5.6 we show experimentally measured values of SSD throughput against various values of Q, N and read/write mixes. As it can be immediately noted from Fig. 5.6a, increasing Q and N improves throughput, as expected. However, if Q is large, further increasing N yields little improvements and vice-versa. One important observation is that, regardless of Q and read/write mix, a segment size of 8–16 is sufficient to achieve most of the gains. Further increasing it yields limited improvements.

With respect to read/write mix, we observe, as expected, that the greatest throughput is achieved in the read-only case, which unfortunately is an unrealistic workload. In that case, the measured throughput approaches the external data rate declared by the manufacturer (4.8 Gbps, as per SSD datasheet). With more realistic read-write mixes (e.g., 10-50% writes), the SSD throughput decreases to about 3.5 Gbps.

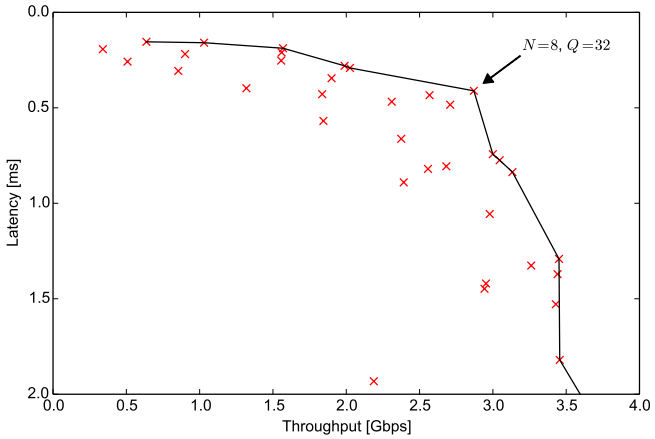


Figure 5.7: SSD latency *vs.* throughput (read/write: 50%/50%)

It is also fundamental to understand how various parameters affect latency. In Fig. 5.7 we depict mean latency and throughput performance for various combinations Q and N in the case of 50% reads and 50% writes. The values of latency reported refer to the time elapsed between the moment an item is passed to an SSD I/O core to the moment the SSD I/O core returns a response.

As it can be noted from the graph, there is an efficient performance frontier. The most important observation is that while it is possible to achieve nearly 3 Gbps of throughput without compromising latency, to achieve a throughput beyond this point heavily penalises latency without substantial throughput gains. We identified that a point providing a suitable latency/throughput tradeoff is for $N = 8$ and $Q = 32$. For this reason, we configure H2C to operate with these parameters and use those for executing all the experiments.

5.4.4 Overall throughput

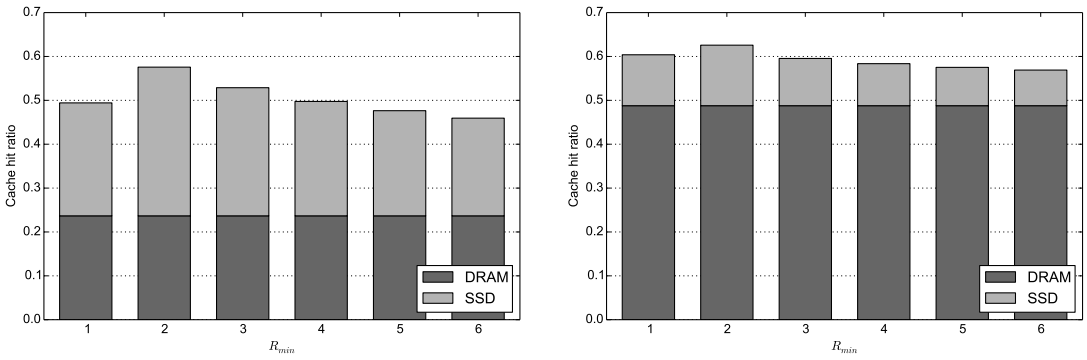
After fine-tuning SSD parameters for a suitable throughput/latency tradeoff, we examine the overall system throughput and report the results in Tab. 5.3. We run experiments using 1 and 2 SSD drives and operating with content objects of 1.5 MB. In both cases, we selected the value of R_{min} that maximises the overall cache throughput (DRAM + SSD) for the scenario considered. The throughput is measured as the Data rate served by H2C averaged over a 60 seconds period.

Table 5.3: Outgoing throughput at various system interfaces using 1 and 2 SSD drives

# SSD	R_{min}	Pkt throughput	Total hits	DRAM hits	SSD hits
1	3	20 Gbps	11.91 Gbps	9.76 Gbps	2.15 Gbps
2	2	20 Gbps	12.44 Gbps	9.76 Gbps	2.68 Gbps

The most striking result is that our design can operate at line speed (20 Gbps) even with only one SSD drive, thus validating the soundness of our design. In both cases considered overall throughput is limited by the network bandwidth. In the first case, however, the SSD drive operates near to the maximum throughput that we measured in isolation (2.15 out of 2.48 Gbps). Differently, in the second case, the two SSD drives operate far from their maximum throughput (2.68 out of 4.96 Gbps). Therefore there is a margin to further increase system throughput if the network bottleneck is removed.

5.4.5 Cache hit ratio



(a) $S = 1.5$ MB

(b) $S = 15$ MB

Figure 5.8: H2C cache hit ratio

We now investigate the hit ratio of H2C, which we break down into DRAM and SSD cache hit ratio in order to understand the contribution of each caching layer. We define DRAM and SSD hit ratios as the ratios of requests served by object located respectively in DRAM and SSD caches over the total number of requests. The total hit ratio is given by the sum of DRAM and the SSD hit ratios. Fig. 5.8 reports the hit ratio as a function of the R_{min} threshold for object sizes S of 1.5 MB and 15 MB.

As expected, we observe that the DRAM hit ratio is not influenced by variations of R_{min}, while the SSD hit ratio determines the total hit ratio shape. Increasing the R_{min} threshold from 1 to 2 improves the SSD hit ratio, as it effectively prevents one-timers (*i.e.,* content objects requested only once) from entering the SSD cache and evicting more popular contents. Further increasing the R_{min} threshold degrades SSD hit ratio. However, selecting greater values of R_{min} has the desirable effect of reducing SSD write load, improving SSD read bandwidth and reducing wear.

Finally, it is also interesting to note that the second layer of the cache significantly contributes to the overall throughput, *i.e.,* from 13% for 1.5 MB content sizes to 33% for 15 MB contents.

5.4.6 SSD throughput

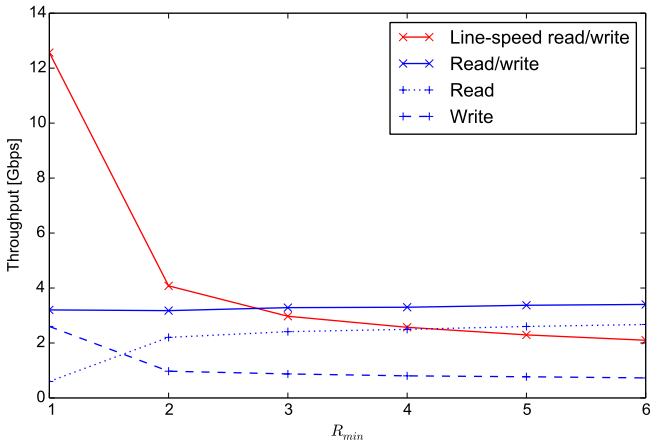


Figure 5.9: SSD read/write throughput *vs.* insertion threshold R_{min}

Finally, we analyse the SSD throughput and how it is impacted by the choice of the R_{min} threshold. To do that, we measure the read-write workload at the SSD layer interface and then we replay it locally to bypass the network bottleneck and evaluate the performance of the SSD layer in isolation.

The results of our experiments, conducted using one single SSD, are depicted in Fig. 5.9. Results for the multi-SSD cases can be easily obtained multiplying throughput values of the single SSD case by the number of drives.

It can be immediately observed that the maximum read/write throughput remains fairly constant independently from the read-write mix, as expected. The most important aspect however is the impact of R_{min} selection on read-write mix and, as a direct consequence, on read and write throughput. In fact, increasing R_{min} from 1 to 2 results in a 3.66x read throughput gain, caused by SSD write reduction triggered by selective SSD insertion. Further increasing R_{min} improves read performance but, as shown in Sec. 5.4.5 also leads to reduced cache hits.

The red line in Fig. 5.9 represents the cumulative read-write throughput that the SSD layer must provide for the system to operate at line speed (20 Gbps). As show in the graph, in this configuration, for $R_{min} \leq 2$, the system throughput is limited by SSD bandwidth, which is encumbered by a heavy write workload. For $R_{min} \geq 3$, the bottleneck is at the network interface.

In Tab. 5.3, we selected $R_{min} = 3$ for experiments with 1 SSD exactly because it is the value maximising cache hits while supporting line-speed operation. In the case of 2 SSD drives, since the available SSD bandwidth is twice as high, only $R_{min} = 2$ is required.

5.5 Related work

5.5.1 Hybrid DRAM-SSD key-value stores

A considerable amount of work addressed the problem of how to efficiently use a combination of DRAM and Flash memories for key-value store applications. However, most of this work focuses on storage rather than caching applications.

The first work addressing the design of two-layer key-value stores is HashCache [15]. However, the specific objective of this work is the design of Web caches running on inexpensive hardware to be used in developing countries using little DRAM and HDD memory. Since DRAM is too expensive for the target use case and not sufficient even to maintain a full index in the form of traditional data structures like hash tables and tries, HashCache explores several index designs trading off DRAM footprint with performance. The most memory-efficient index proposed uses no DRAM at all. However, this memory efficiency comes at a cost of very low throughput performance.

FAWN [7] also proposed a key-value store using DRAM and SSD. However, the objective of this design is primarily energy efficiency and therefore all design decisions have been geared towards this objective, including using low-power CPUs and primarily SSD.

CLAM [5] uses DRAM as a write buffer for SSD, where items are inserted in very large batches in a log structure. Lookups for items stored in SSD are executed by querying a set of Bloom filters kept in DRAM. The use of Bloom filters makes CLAM fairly memory-efficient and a lookup requires only one SSD read. Although CLAM supports caching semantics, it does not support filtering of items before insertions, hence suffers of all the limitations identified in Sec. 5.2.2. Also, except for items temporarily buffered in DRAM, all contents, whether popular or not, are stored in SSD, which bounds the maximum achievable throughput to the limited bandwidth offered by SSDs.

SkimpyStash [50] supports only storage semantics and targets applications requiring very large amount of data, such as storage deduplication. As such, its design focuses on maximising memory efficiency. Its index requires only 1 ± 0.5 B/key as opposed to the 4 B/key required by CLAM. However, this improvement comes at a cost of high SSD read amplification (5 reads/lookup as opposed to 1 read/lookup of CLAM).

SILT [119] shares the same of objectives of SkimpyStash, but improves upon it in terms of both memory efficiency and read amplifications. Its index has a memory footprint of just 0.4B/key and requires 1 read/lookup in SSD. This memory efficiency and low read amplification comes however at a cost of requiring periodic reorganisation of data on SSD, hence causing high computational overhead. It also does not support caching semantics.

FlashStore [49] uses DRAM and Flash memory as a read/write-through cache for HDD-based storage systems. Although it specifically focuses on a caching use case, it cannot silently evict written items without inserting them to HDD.

Finally `fatcache` [169] is Twitter's implementation of `memcached` [130] supporting SSD storage. Similarly to other designs, it uses DRAM only as an index and stores content items in SSD in a log-structured store. Despite being the only DRAM/SSD key-value store specifically targeting caching, it still shares the limitations of other designs, most notably the insertion of all items without any prefiltering and the use of DRAM only as index, both of which detriment throughput performance.

All these designs have fundamental limitations that make them unsuitable for the content distribution use case. First of all, most of them do not support caching semantics. Those that do also support storage semantics and above all target applications requiring the storage of a very large amount of entries. As

such their design is mostly geared towards reducing the size of the index and read/write amplifications without considering other metrics of primary importance for caching systems such as cache hit ratio.

5.5.2 ICN content store design

Another stream of work related to our contribution concerns the design of content stores specifically targeting ICN routers.

Perino and Varvello [137] made seminal contributions by carrying out a systematic review of design options, evaluating feasibility and cost-effectiveness, without however providing a complete design.

The first design of a line-speed content store was provided by So *et al.* [155]. They presented a complete router design running on a service card for high-end IP routers, albeit supporting only DRAM memory. In their design they propose an optimised hash table jointly supporting PIT and CS operations. The same authors, later presented a high level design for supporting SSD-based caching as a separate module [154]. However, their contribution only consists in general architectural considerations with a performance evaluation limited to SSD I/O.

Rossini *et al.* [148] presented a design, without implementation, of a two-layer DRAM/SSD ICN router. However, their design focuses on the specific use case of video traffic and assumes proactive caching (*i.e.,* caches are prepopulated during off-peak periods). As such their design disregards all the complexity of concurrent read/write operations that are crucial for high throughput performance. H2C design uses some concepts proposed in [148], such as batched SSD I/O operations. However our contribution widely extends this work by providing a complete practical design supporting any type of traffic with concurrent reads and writes.

Finally, Thomas *et al.* [164] also investigated design issues of two-layer ICN routers, although they focused on the specific case of SRAM and DRAM memories. Their design employs techniques to reduce the impact of large and unpopular content objects on cache hit performance. These issues are also implicitly addressed by our design by caching at a segment granularity (as opposed to a content object granularity) and selectively inserting items in SSD.

To the best of our knowledge, H2C is the first implementation of a complete content router with a DRAM/SSD cache operating at line speed.

5.6 Conclusion

Read-through caching nodes are a fundamental building block of a variety of networked systems such as CDNs, ICNs [89], [183] and redundancy elimination applications [6]. Despite their importance, their design and implementation have not received adequate attention from the research community.

In this chapter we improve the state of the art with two main contributions. First, we present a set of general design principles applicable to a wide variety of applications. Second, we apply these principles and complement them with more specific optimisation techniques to design H2C, a packet-level cache that we evaluate, as use case, as content store of an ICN router.

Overall, we proposed a number of techniques substantially departing from previous work. These include:

- A lightweight cache insertion filtering technique improving cache hit ratio and SSD read throughput
- Organisation of consecutive chunks into segment to improve SSD read/write throughput and reduce data structure memory overhead.
- System-wide performance optimisation such as batching, prefetching, zero-copy operations and exploitation of hardware parallelism.

Experimental results with realistic traffic traces show H2C sustains multi-Gbps throughput on inexpensive commodity hardware.

Chapter 6

Conclusions and future work

6.1 Summary

Content distribution is the use case currently accounting for the majority of Internet traffic and is forecast to constantly increase over the coming years. This trend is challenging the scalability and reliability of the global Internet, partly owing to the fact that the latter was designed having in mind host-to-host communications as its main use case rather than user-to-content access.

It is generally accepted that the most effective solution to this problem is through the deployment of globally distributed caching systems as the way to localise content distribution traffic and preserve stability. This solution has already been deployed in practice with the advent of Content Delivery Networks. In addition, ubiquitous caching is also a fundamental feature of many future Internet architectures. However, with the growing amount of content distribution traffic, caching systems are expected to grow accordingly in terms of both footprint and traffic carried. As a result, their cost and complexity are also expected to grow.

This thesis addressed the problem of scalability and cost-effectiveness of distributed caching systems. It achieved this objective by making two main contributions tackling complementary issues in distributed caching systems. These are the design of efficient, load-balanced and predictable systems of caches and the design of efficient and cost-effective caching nodes that are part of a network of caches.

The first contribution, presented in Chapter 4, is the design of a caching framework specifically targeting the case of caches controlled by a network operator. This framework provides a set of key advantages over state-of-the-art techniques. It provides a 2x improvement in cache hit ratio and a 19-33% reduction in load imbalance while still yielding comparable latency. It also provides important qualitative advantages, such as robustness against traffic peaks and flash crowds, predictable performance and availability of knobs to fine-tune the performance.

The second contribution, presented in Chapter 5, is the design and implementation of a high-speed caching node that can achieve good cache hit ratio and 20 Gbps line-speed operations based on inexpensive commodity hardware. To the best of our knowledge, this is the first complete implementation of a content router operating at line speed with a DRAM/SSD cache. In addition, many design principles devised here are more generally applicable to other use cases.

Both contributions rely on common theoretical findings which have been presented in Chapter 3.

These results shed light on the performance of sharded caching systems particularly regarding load balancing and caching efficiency. Given the ubiquitous adoption of sharding techniques in computer systems, the findings presented here have wider applicability than the scope of this thesis.

In summary, this thesis provided a well rounded set of contributions addressing the problem of building scalable and cost-effective caching systems at two different levels, *i.e.,* the design of specific nodes and their interconnection. These contributions are rooted in solid theoretical foundations that motivated their design. We think that the contributions of this thesis advance significantly the state of the art in distributed caching system.

6.2 Future work

Future research directions can be summarised as follows.

Extension of sharding performance models. The theoretical characterisation of sharded caching systems presented in Chapter 3 has been carried out with the main objective of providing theoretical foundations motivating decisions made in the design contributions of Chapters 4 and 5. However, as already mentioned, sharding techniques are ubiquitously utilised in a variety of applications and the modelling contributions provided can be easily applied to the design of many operational systems. While the work presented in this thesis already covers important aspects, such models can be extended to apply to more general cases and to take into account further performance metrics.

One possible direction for extension consists in addressing the case of heterogeneous shards. Current models assume that all shards have identical cache size and processing capabilities. However, this may not be the case in reality, because as a result of incremental system upgrades, sharded systems may comprise entities with different hardware configuration. Generalising cache hit ratio and load balancing results for the case of heterogeneous processing and cache size capabilities would make the model suitable to address this case as well.

Another possible direction concerns the generalisation of the analysis of load balancing performance in the presence of frontend caches presented in Sec. 3.3.5. While the work presented in this thesis assumes perfect caching, there is practical interest in generalising these results to the case of recency-based replacement policies such as LRU, FIFO and derivatives. Early results from numerical evaluations (not presented in this thesis) suggest that the findings presented here hold also for a larger set of replacement policies. However, further investigation is required to validate this.

Finally, and probably most importantly, there would be great benefit in extending the current work to understand how load balancing and caching performance effectively impact more concrete metrics, such as latency and throughput. This could be achieved by making specific assumptions about the operation of each shard and applying concepts from queueing theory.

Investigate optimal configuration of multiple replicas caching. One extension to the base caching framework presented in Chapter 4 is the support for multiple content replicas, as explained in Sec.

4.3.3. By supporting multiple content replicas, a hash-routed caching system operates as a hybrid between pure hash-routing and on-path caching, depending on the number of content replicas in the system. At one extreme, where each content is replicated exactly once, the system operates like a pure hash-routing scheme. At the other extreme, where each content is replicated as many times as the number of caching nodes, the system operate as a pure on-path caching scheme.

A multiple-replica hash-routing scheme is in practice a generalised caching framework encompassing on-path and hash-routing caching. As a result, it could give operators great flexibility in allowing them to fine-tune performance by trading off cache hit ratio, latency, load balancing and scalability through careful selection of the degree of replication, the assignment of content replicas and the routing strategy. Unfortunately, the optimal parameter configuration in the presence of multiple replicas is difficult. However, the results presented in Sec. 4.6.5 show that even with a simple replica placement based on network clustering it is possible to achieve good results, which is very encouraging.

A promising future research direction then consists in investigating how to optimise the configuration of the distributed caching framework of Chapter 4 in the presence of multiple replicas.

Generalisation of caching node design. The caching node implementation presented in Chapter 5 has been primarily designed to operate as a Content Store for an ICN router and this objective has dictated a number of design decisions. However, most of the design principles adopted are more widely applicable and could be exploited to improve the design of other applications. Therefore, one interesting direction for future work concerns the generalisation of H2C implementation to support a larger number of applications. Examples of applications that could be targeted include HTTP proxies and caching key-value store applications, such as `memcached` [130].

Also, another possible extension concerns the decoupling of H2C implementation from the other stages of packet processing of an ICN router (*i.e.,* FIB and PIT lookups). This would improve the portability of H2C and would make it a suitable drop-in replacement able to operate with other ICN router implementations. A possible approach to achieve this objective would be to adapt the H2C implementation to export a well-defined interface to the rest of the system, such as a filesystem API, using tools like the Filesystem in Userspace (FUSE). However, this decoupling may degrade performance as a result of the introduced overhead.

Appendix A

Simplifying the configuration of controlled network experiments

A.1 Introduction

The setup of a realistic scenario for a controlled network experiment, whether in a simulated or emulated environment, is usually a lengthy and delicate process comprising various tasks.

The first task consists in selecting a suitable network topology. Such a topology can either be parsed from datasets of inferred topologies, such as RocketFuel [157] or the CAIDA AS relationships dataset [27] or synthetically generated according to various models, such as [4], [18], [26], [54] or [179]. Alternatively, it is also possible to use canonical topologies such as stars, rings or dumbbells.

Second, after selecting the topology, it is necessary to configure it with all required parameters to be used in the target simulator or emulator. These include link delays, capacities, weights, buffer sizes and configuration properties of all protocol stacks.

Third, it is necessary to assign a traffic matrix to the topology or decide how the traffic will be modelled, such as deciding on the number of concurrent flows, their origin and destination and their characteristics.

Finally, all this configuration has to be implemented in the target environment before the experiment can be run.

The execution of all these tasks is cumbersome and error-prone, since there are no publicly available tools automating the entire process. In fact, although there are tools taking care of some of the tasks, such as the parsing or the generation of topologies, they do not support the entire setup chain and are generally bound to a specific simulator or emulator. As a result, a user is required to integrate heterogeneous software components or implement a complete experiment scenario from scratch.

Apart from possibly requiring a considerable amount of time, this process can also lead to an increased amount of mistakes affecting the reliability of results. In fact, the lack of a framework for automating experiment setup may lead users to configure network and traffic characteristics using unrealistic models. In addition, even if appropriate models are selected, defects may be introduced in their actual implementation. For all these reasons, it is highly desirable having a tool supporting the entire setup chain.

To address these issues, this chapter presents the design and implementation of a toolchain for easily executing all the tasks listed above. It allows users to parse topologies from various datasets or from other generators, as well as generating them according to the most common models. These topologies can then be configured with all required parameters and matched with appropriate traffic matrices or traffic source configurations. A fully configured experiment scenario can be exported to a set of XML files which can then be imported by the desired experiment environment. The toolchain provides adapters for ns-2 [132], ns-3 [77], Omnet++ [174], Mininet [111], AutoNetKit [105], jFed [95] as well as generic Java, C++ and Python APIs to enable an easy integration with other simulators or emulators. In particular, by providing generic APIs for the most common programming languages, we hope to contribute to increase the reliability and reduce the setup complexity of experiments run with custom-built simulators, which are very common.

The methods provided by the toolchain for generating and configuring network topologies are commonly used in literature, with the exception of those used for link capacity assignment. In fact, for this task, apart from providing commonly adopted models, we devised and implemented novel algorithms which provide a more realistic link capacity assignment than state-of-the-art methods. This chapter presents these new models and demonstrate their effectiveness by evaluating their performance on a number of real network topologies.

This toolchain was originally named *Fast Network Simulation Setup (FNSS)* and is publicly available as open-source software [61]. However, since it was first made publicly available evolved to support also emulated environments in addition to simulators.

The remainder of this chapter is organised as follows. Sec. A.2 describes the FNSS toolchain by explaining its architecture and design and illustrating its features. Sec. A.3 introduces novel link capacity assignment algorithms and evaluates their performance. Sec. A.4 presents a complete example of how FNSS can be used. Sec. A.5 provides an overview of the related work. Finally, Sec. A.6 summarises the contribution of this chapter and presents the conclusions.

A.2 The FNSS toolchain

A.2.1 Architecture and design

The FNSS toolchain comprises a core library, a C++ API, a Java API and an ns-3 API.

The core library implements all functionalities required to parse or generate topologies, configure them, generate traffic matrices and event schedules and save them as XML files. It also comprises all code required to export scenarios to ns-2, Omnet++, Mininet, AutoNetKit and jFed. The C++, Java and ns-3 APIs are separate components written in a different programming language from the core library. They can parse the XML files generated by the core library and convert them to objects which can be used by the target experiment environment.

The core library is entirely written in Python, with some of its functionalities built on top of the NetworkX library [76]. We selected Python as the programming language for two main reasons. First, its high-level programming constructs would allow FNSS users to generate complex network simulation

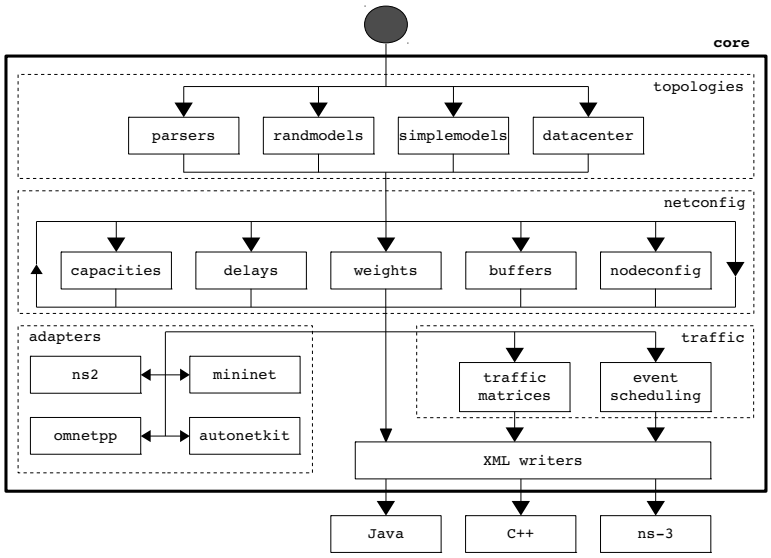


Figure A.1: FNSS workflow

scenarios with a few lines of code. Second, the availability of NetworkX, a well-designed and actively-maintained library for graph manipulation and visualisation, provides useful tools to manipulate topologies created with FNSS.

As depicted in Fig. A.1, which shows the workflow on which FNSS design is based, the core library has been implemented following a modular architecture. All core functionalities are implemented in 15 modules, contained in 4 packages, namely `topologies`, `netconfig`, `traffic` and `adapters`. The code functionalities are allocated to the various packages as follows:

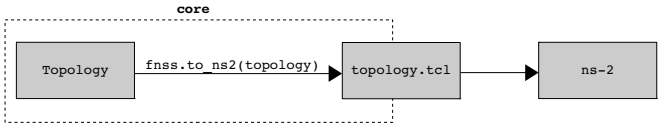
- **`topologies`**: contains all functions allowing users to parse or synthetically generate network topologies and export them to an XML file.
- **`netconfig`**: contains all functions required to assign configuration parameters to network topologies, precisely: link capacities, link weights, link delays, buffer sizes, protocol stacks and applications.
- **`traffic`**: contains functions for synthetically generating traffic matrices and event schedules and to export them to XML files.
- **`adapters`**: contains functions to export scenarios generated with FNSS to a number of target environments.

In the core library, the entities required to represent an experiment scenario are modelled with objects belonging to three categories:

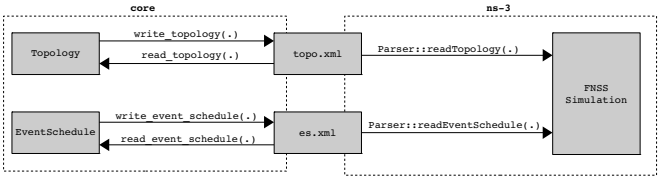
- **Topologies:** they represent fully configured network topologies. There are three different topology classes: `Topology`, `DirectedTopology` and `DatacenterTopology`.
- **Traffic matrices:** they represent traffic matrices. A static traffic matrix is represented by a

`TrafficMatrix` object, while a dynamic traffic matrix is represented by `TrafficMatrixSequence` object.

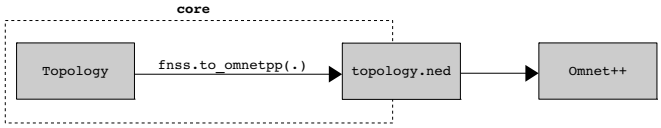
- **Event schedules:** they represent a schedule of events labelled with an execution time. In the core library, the class modelling schedules of events is called `EventSchedule`



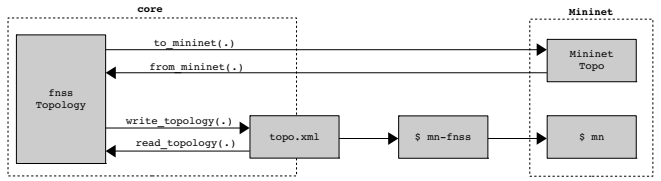
(a) ns-2



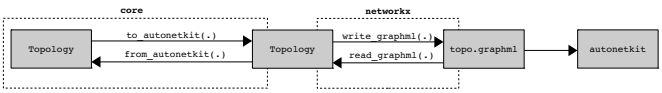
(b) ns-3



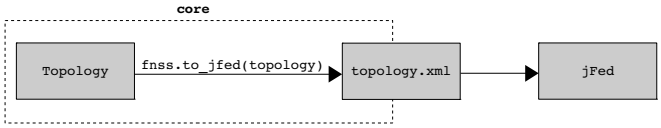
(c) Omnet++



(d) Mininet



(e) AutoNetKit



(f) jFed

Figure A.2: FNSS integration strategies

All these objects can be serialised and saved in XML files, which can then be parsed by the adapter for the desired target environment. Various simulators and emulators are currently supported by FNSS. However, the strategies used to support them are different (see Fig. A.2). Supported environments are integrated as follows:

- **ns-2:** The ns-2 adapter is a Python module of the core library which takes as input a topology (in

the form of either an XML file or an FNSS Python object) and, using a templating engine, converts it into a Tcl script containing all the code needed to set up the topology in ns-2.

- **ns-3:** The ns-3 adapter comprises a set of C++ classes which parse topology and event schedule XML files, deploy the configuration specified in the topology file and schedule the execution of the tasks specified in the event schedule in the ns-3 environment.

- **Omnet++:** The Omnet++ adapter, similarly to the ns-2 adapter, is a Python module of the core library which takes as input a topology (in the form of either an XML file or an FNSS Python object) and, using a templating engine, converts it into a NED script containing a description of the topology in a format readable by Omnet++.

- **Mininet:** The Mininet adapter is a Python module of the core library capable of converting an FNSS `Topology` object into a Mininet `Topo` object, which is a Python object modelling topologies used by Mininet. This adapter also allows to convert Mininet topologies back to FNSS. FNSS also includes a script (`mn-fnss`) which launches the Mininet console (`mn`) initialised with an FNSS topology.

- **AutoNetKit:** The AutoNetKit adapter, similarly to the Mininet adapter, converts FNSS topologies into NetworkX `Graph` objects with node and link labels compatible with AutoNetKit expected format and viceversa.

- **jFed:** The jFed adapter converts FNSS `Topology` objects into XML files compatible with jFed.

- **Generic Java, C++ or Python simulator:** The generic APIs provide capabilities to parse the XML files of topologies, traffic matrices or event schedules and convert them into objects for the target language.

In the remainder of this section, we will describe in more detail all functionalities provided by FNSS.

A.2.2 Network topologies

FNSS allows users to create topologies in a variety of different ways:

- **Import a topology from a dataset:** topologies can be parsed from the RocketFuel ISP maps [157], the CAIDA AS relationships dataset [27], the Topology zoo dataset [106] and the Abilene network topology [1].

- **Import a topology from other topology generators:** FNSS supports the import of topologies generated using BRITE [127], Inet [181] and aSHIIP [165].

- **Generate a synthetic random topology:** the models supported are Barabási-Albert [18], extended Barabási-Albert [4], Erdős-Rényi [54], Waxman [179] and Generalised Linear Preference (GLP) [26].

- **Generate a datacenter topology:** the models supported are two- and three-tier [21], fat tree [3] and BCube [75].

- **Generate a simple topology:** FNSS supports the generation of the following canonical topologies: k-ary tree, dumbbell, line, star, ring and full mesh.

A parsed or generated topology is an instance of either the `Topology`, `DirectedTopology` or `DatacenterTopology` classes, all of which extend the NetworkX `Graph` class. This design decision makes it possible to directly use the graph algorithms provided by the NetworkX library on such topologies.

In the following section we explain in more detail the synthetic and datacenter topology models provided by FNSS and discuss how to select appropriate model parameters.

A.2.2.1 Synthetic models

FNSS can create synthetic topologies according to the most common models. These are:

- **Erdős-Rényi [54]:** it generates simple random topologies $G(n, p)$, where n is the number of nodes and p is the probability that a pair of nodes is connected by an edge.

- **Waxman [179]:** it generates topologies in which the probability that two nodes are connected by an edge depends on their distance. Such probability is equal to:

$$p(u, v) = \alpha e^{-d(u,v)/\beta L} \tag{A.1}$$

  where $\alpha, \beta \in (0, 1]$ are required model parameters and $L > 0$ is the maximum distance between two nodes. By increasing the value of α raises the edge density, while decreasing the value of β makes the probability function decay faster, resulting in a greater density of short edges in comparison to long ones.

  In FNSS, topologies generated with this model have edges annotated with their Euclidean distance, which makes it possible to later assign link weights and delays accordingly.

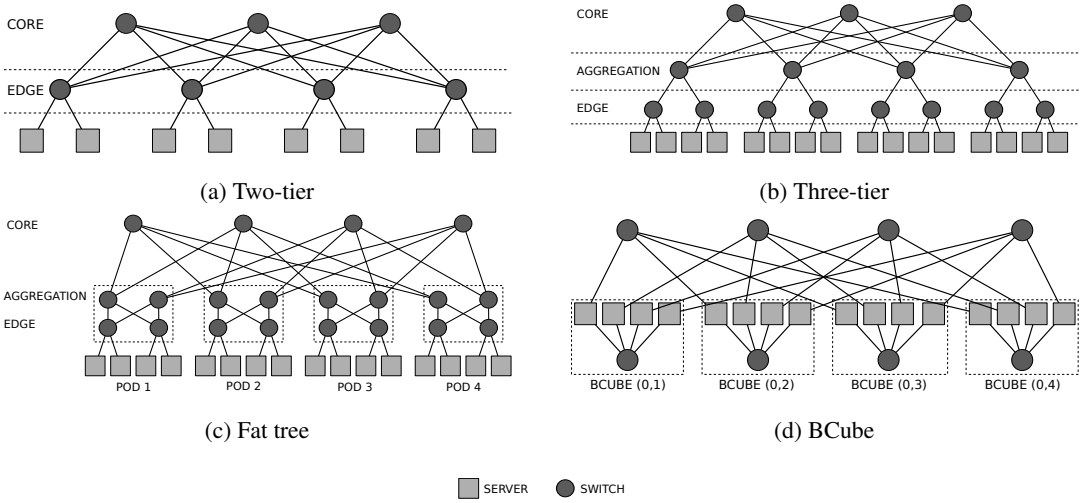
- **Barabási-Albert [18]:** it generates scale-free networks whose node degrees are power-law distributed. Such topologies are created by adding nodes according to a *preferential attachment* model, whereby new nodes joining the network are more likely to attach to nodes with a greater degree. In such a topology, the probability that a node has degree k is k^{-3}, irrespective of its size and of the value of model parameters. Topologies generated with this model, as well as with other models that provide scale-free graphs, are generally well-suited to model AS-level Internet topology, since, as demonstrated by [56], Internet topology exhibits power-law characteristics.

- **Extended Barabási-Albert [4]:** it is an extension of the Barabási-Albert model that takes into account the presence of local random events such as the addition of nodes or links or the rewiring of existing links in the construction of the topology. The result is still a scale-free topology. However the exponent of the node degree distribution varies.

- **Generalised Linear Preference (GLP) [26]:** it is another model capable of generating scale-free networks in a similar manner to the two models previously presented. It is a small extension of

the extended Barabási-Albert model that comprises an additional parameter $\beta \in (-\infty, 1)$ which allows users to fine-tune the intensity of the preferential attachment. Decreasing the value of β reduces the preference given to high degree nodes for attachment.

A.2.2.2 Datacenter models



(a) Two-tier (b) Three-tier

(c) Fat tree (d) BCube

Figure A.3: Models of datacenter topologies provided by FNSS

The dramatic popularity increase experienced by cloud computing has triggered a considerable amount of research focusing on datacenter issues. As a result, novel topologies have been proposed in order to achieve specific objectives, such as performance improvement, cost reduction or simplification of wiring In order to address the increasing need for tools to evaluate datacenters, FNSS has been equipped with functionalities for generating the most common datacenter topologies. The topologies, depicted in Fig. A.3, are:

- **Two- and three-tier [21]**: these are arguably the most common datacenter topologies. They are characterised by the fact that switches are organised in two tiers (core and edge) or three tiers (core, aggregation and edge). Three-tier topologies are generally capable of supporting hundreds of thousands of servers, while two-tier topologies can only support up to five to eight thousand servers.

- **Fat tree [3]:** it is a recently proposed topology, designed to build large-scale datacenter using only commodity k-port switches appropriately interconnected in a way similar to a Clos network [43].

- **BCube [75]:** it is a topology specifically designed for shipping-container based, modular datacenters. In this model, switches are never directly connected to each others and servers also accomplish packet forwarding functions.

A.2.3 Topology configuration

A.2.3.1 Link capacities

As briefly mentioned at the beginning of this chapter, with respect to link capacity assignment, FNSS implements both commonly used and newly proposed methods.

The common methods implemented by FNSS are:

- **Constant capacity**: all links are assigned a fixed user-defined capacity.
- **Manual assignment**: the user can assign specific capacities to selected links.
- **Random capacity assignment**: the user can assign capacities randomly. To use this model, the user needs to specify a set of allowed capacities and the probability of each capacity being assigned to a link. Apart from allowing users to manually specify the probability density function, FNSS provides utility methods for assigning capacities according to uniform, power-law and Zipf-Mandelbrot distributions.

The newly proposed models allow users to assign link capacities proportionally to a number of topological centrality metrics. These models are formally introduced and evaluated in Sec. A.3.

A.2.3.2 Link delays

Link delays can be assigned in three different ways:

- **Geo-distance**: if the topology has been parsed from a dataset providing the geographical coordinates of the nodes, it is possible to assign link delays proportionally to the estimated length of the link, calculated as the Euclidean distance between the two endpoints. The user can specify the desired value of specific propagation delay to model different types of transmission media.
- **Constant delay to all links**: all links are assigned the same delay.
- **Manual delay assignment**: the user can manually select a link or a set of links and assign them a specific delay.

A.2.3.3 Link weights

Link weights can be assigned in four different ways:

- **Proportionally to the inverse of link capacity**: the weight assigned to each link is proportional to the inverse of its capacity.
- **Proportionally to link delay**: the weight assigned to each link is proportional to its delay.
- **Constant weight to all links**: all links are assigned the same weight (default is 1).
- **Manual weight assignment**: the user can manually select a link or a set of links and assign them a specific weight.

In addition, if the topology has been parsed from a dataset already providing link weight information, such as the Abilene topology [1], weights are automatically assigned.

A.2.3.4 Buffer sizes

Buffer sizes can be assigned in four different ways:

- **Equal to network bandwidth-delay product:** This method follows the rule of thumb, originally proposed by [176], that the buffer sizes of Internet routers should be set to the average RTT experienced by TCP flows traversing them multiplied by the link bandwidth. In FNSS, this is calculated as follows:

  1. Identify all shortest paths traversing the specific buffer.
  2. Calculate the RTT for each of these paths by summing the propagation delay of each link in the path.
  3. Average the RTT and multiply it by the capacity of the link associated to the buffer.

- **Proportionally to link capacity**: In case link propagation delays are not known or are set to 0, it is possible to assign a buffer size equal to $k \times C$ where C is the capacity of the link and k is a constant.

- **Constant buffer size**: All buffers in the topology are assigned a fixed size.

- **Manual assignment**: A user can manually assign specific sizes to a single buffer or a set of buffers.

A.2.3.5 Protocol stacks and applications

As already mentioned above, FNSS provides the capability to preconfigure protocol stacks and applications and deploy them on selected nodes of a topology. Each types of stacks and applications are identified by a specific name (e.g. `http_server`) and a set of key-value attributes stored in a dictionary. For example, an HTTP server application may have the following properties:

```
server_props = {
    'port': 80,
    'max_threads': 200,
    'avg_obj_size': '150KB'
    'conn_keep_alive': True
}
```

The application thus defined is appended to the selected node(s) of the topology object. When the topology is saved to an XML file, all stack and application information is saved as well so that it can be easily imported by the target simulator or emulator.

A.2.4 Event scheduling

Another feature of FNSS is the ability to produce event schedules and export them to an XML file. An event schedule is represented in the core library by an object of the `EventSchedule` class. Such a schedule is a sorted list of events labelled with an execution time. Each event, similarly to applications and protocol stacks, is modelled as a dictionary of key-value attributes.

For example, an HTTP request event could be represented by the following set of attributes:

```
event = {
    'client_ip':   '192.168.1.24'
    'proxy':       '192.168.1.100:8080'
    'method':      'GET'
    'url':         'http://www.ucl.ac.uk/'
    'User-Agent':  'fnss-client'
    'Connection':  'keep-alive'
}
```

A.2.5 Traffic matrices

FNSS is capable of synthetically generating traffic matrices with given statistical characteristic which can be used for network experiments.

A Traffic Matrix (TM) is a representation of aggregate traffic volumes being carried by a network at a specific time interval, whose elements T_{ij} represent the average traffic volume from ingress node i to egress node j.

There exist two types of traffic matrices: *static* or *dynamic*. A static traffic matrix reports the traffic volumes collected at a single point in time. Differently, a dynamic matrix contains a sequence of traffic volumes collected at different times.

In a communication network, traffic volumes follow diurnal patterns [156]. Therefore, to appropriately model network traffic over long time spans, it is recommendable to adopt dynamic traffic matrices which are cyclostationary over a period of 24 hours. However, it has been observed [133] that over shorter timescales, generally under one hour, traffic variations can be accurately modelled using just stationary dynamic traffic matrices. Static traffic matrices, instead, can only be reliably used to model network traffic at a single point in time.

From a mathematical perspective, *cyclostationary*, *stationary* and *static* traffic matrices can be represented as follows.

A cyclostationary traffic matrix can be represented as:

$$T_{ij}(t) = X_{ij}(t) + W_{ij}(t) \tag{A.2}$$

where $X_{ij}(t)$ is the mean traffic volume from i to j, periodic of period $T = 24h$ and $W_{ij}(t)$ is a zero-mean random variable modelling random fluctuations of traffic volumes.

A stationary traffic matrix can be represented as:

$$T_{ij}(t) = X_{ij} + W_{ij}(t) \tag{A.3}$$

where X_{ij}, unlike the cyclostationary model, is time-invariant.

Finally, a static traffic matrix can be represented as:

$$T_{ij}(t = t_0) = X_{ij} \tag{A.4}$$

where, since the matrix comprises a single set of traffic volumes, there are no volume fluctuations over time.

FNSS is capable of synthetically generating both static, stationary and cyclostationary traffic matrices. This is realised following the process proposed by Nucci *et al.* in [133], which comprises the four following steps.

First, we randomly generate the mean rates for all flows between each ingress and egress node X_{ij} as realisations of a lognormal random variable $ln\mathcal{N}(\mu, \sigma^2)$. At this stage we simply generate as many mean values of traffic volume as the number of Origin-Destination (OD) pairs in the network but we do not map them to specific OD pairs yet.

Second, in case a dynamic (either stationary or cyclostationary) traffic matrix is desired, we generate random fluctuations for each OD pair. These fluctuations are realisations of a zero-mean normal random variable. As shown in [133] and [28], the standard deviation of such fluctuations σ and the mean traffic volumes are related by the following power law:

$$\overline{x}_{i,j}(t) = \psi\sigma_{ij}^{\gamma} \tag{A.5}$$

Nucci *et al.* [133] reported that in Sprint Europe and Abilene [1], this assumption about power-law relationship holds. In particular, the values that best fit the distribution are ($\gamma = 0.8$, $log\psi = -0.33$) for Sprint and ($\gamma = 0.93$, $log\psi = -0.31$) for Abilene.

Third, in case a cyclostationary traffic matrix is required, traffic volumes are multiplied by a one-mean $\sin$ function oscillating between $1 - \delta$ and $1 + \delta$ in order to simulate diurnal traffic oscillations.

Fourth, we assign traffic volumes to the target topology. This assignment is executed using the *Ranking Metrics Heuristic* proposed by [133]. This method comprises the following steps:

1. Sort all volumes X_{ij} in decreasing order
2. Sort all OD pairs in decreasing order of metric m_1:

   $$m_1(n_1, n_2) = \min\left(F_{out}(n_1), F_{in}(n_2)\right) \tag{A.6}$$

   where F_{out} and F_{in} respectively the fan-out and fan-in capacity of a node.
3. If a tie occurs, subsort OD pairs in decreasing order of metric m_2 where:

   $$m_2(n_1, n_2) = \min\left(\text{outdeg}(n_1), \text{indeg}(n_2)\right) \tag{A.7}$$

4. If another tie occurs, subsort OD pairs in decreasing order of metric m_3 where:

   $$m_3(n_1, n_2) = \frac{1}{\max\left(\text{NFUR}(n_1), \text{NFUR}(n_2)\right)} \tag{A.8}$$

   and NFUR stands for *Number of Flow Under Failure* and corresponds to the maximum number of shortest paths traversing a node if one random link of the network is removed.
5. Map sorted traffic volumes to sorted OD pairs

Finally, we calculate the utilisation of each link assuming that shortest path routing is used and linearly scale traffic volumes to match the expected maximum link utilisation.

In FNSS implementation, a static traffic matrix can be generated by invoking the function `static_traffic_matrix (topology, mean, stddev, max_u)` where `topology` is the topology for which the matrix is generated, `mean` and `stddev` are respectively the values of μ and σ used to generate the mean flows and `max_u` is the target maximum link utilisation used for scaling traffic volumes. The function to be invoked for generating dynamic (stationary) matrices is `stationary_traffic_matrix(topology, mean, stddev, gamma, log_psi, n, max_u)` where `n` is the number of time intervals and `gamma` and `log_psi` are respectively the values of γ and $log\psi$ used to derive the standard deviation of random fluctuations. Finally, a cyclostationary traffic matrix can be generated by invoking the function `sin_cyclostationary _traffic_matrix(topology, mean, stddev, gamma, log_psi, delta, n, periods, max_u)` where `delta` is the value δ of the sin function, `n` is the number of traffic samples per period and `periods` in the number of periods spanned by the matrix.

A.3 Link capacity estimation

While a large amount of work is available in literature regarding the modelling ([4], [18], [26], [179]) or the inference ([51], [73], [157]) of Internet topologies, almost no work exists on inference and modelling of link capacity distribution.

With respect to link capacity inference, although a number of methods for inferring link capacities have been proposed ([30], [88]), their practical applicability is limited because it is hard to accurately measure packet timing or congest network links to the point required for measuring link capacity before the ISP reacts [182].

Differently, with respect to link capacity modelling, to the best of our knowledge, the only work investigating properties of link capacity distribution in an ISP network is [80]. In this paper, the authors show that, in the backbone network of the Internet Initiative Japan (IIJ) ISP, the number of links with the same capacity follows a power law. The authors speculate that this relation could be due to the fact that by allocating link capacities according to a power law would yield better network throughput performance in comparison to other distributions. These results, however, cannot be considered conclusive because they have been only validated on a single topology.

The lack of commonly accepted models for link capacity distribution has resulted in various heterogeneous methodologies being adopted in practice.

The BRITE topology generator, for example, can assign either a constant user-defined capacity to all links or assign capacities randomly. In the latter case, the user can specify a minimum and a maximum capacity BW_{min} and BW_{max} and a distribution (uniform, exponential or Pareto) and the BRITE tool assigns to each link a random capacity $C = [BW_{min}, BW_{max}]$ according to the selected distribution. This method, however, cannot generate realistic capacity assignments. In fact, in this model link capacities can take any real value between minimum and maximum capacities while links in real networks can only have a discrete number of different capacities.

Another common approach ([140], [182]) is to select a set of discrete capacities and repeat experiments using different constant capacities or assign heterogeneous capacities according to various

distributions and evaluate the sensitivity of results.

The use of this large variety of not validated models can have a negative impact on the reliability of network experiments whose results are sensitive to capacity assignment. To address this issue, in the following sections, we analyse the problem of modelling link capacity distributions and propose novel methods for modelling the link capacity assignments of ISP backbone networks. Our evaluation, presented in Sec. A.3.2, shows that our methods provide more realistic assignments than commonly used random assignments.

A.3.1 Proposed solution

The problem we are addressing in this section is the following. Let $G(E, V)$ be a graph with vertices $V = \{v_1, v_2, ...v_n\}$ and edges $E = \{e_1, e_2, ..., e_m\}$ representing the topology of an ISP backbone network whose edges have capacities belonging to the set $C = \{c_1, c_2, ..., c_p\}$, with $|C| \ll |E|$, assigned according to $f_c : E \rightarrow C$. The objective is to find the most accurate estimate $\hat{f}_c$ of f_c, assuming that no additional information about the network apart from G and C is available.

Our intuition is that, in ISP backbone networks, capacities are not allocated to links randomly but according to various economic, technological and traffic constraints, including the *importance* of the links, which are also related to the *importance* of the nodes they connect. For this reason, we speculate that there should be some correlation between the amount of capacity assigned to a specific link and various metrics capturing the importance of the link itself or the link's endpoints. Our argument is therefore that important links are more likely to have greater capacities.

Based on this assumption, the solution proposed here is to identify a set of metrics m capable of capturing links importance and to assign link capacities c which are proportional to the value of such metric. As a result, the capacity assigned to each link (u, v) is:

$$c(u, v) = c_i \in C : b_{i-1} \leq \tilde{c}(u, v) < b_i \tag{A.9}$$

where:

$$\tilde{c}(u, v) = b_0 + (b_{|C|} - b_0)\frac{m(u, v) - \min(m)}{\max(m) - \min(m)} \tag{A.10}$$

and:

$$b_i = \begin{cases} c_1 - \frac{c_2 - c_1}{2}, & \text{if } i = 0 \\ \frac{c_i + c_{i+1}}{2}, & \text{if } 1 \leq i \leq |C|-1 \\ c_{|C|} + \frac{c_{|C|} - c_{|C|-1}}{2}, & \text{if } i = |C| \end{cases} \tag{A.11}$$

In our analysis, we focus on three specific metrics m to capture the importance of edges in a graph. Such metrics are the *edge betweenness centrality*, the *degree centrality gravity* and the *communicability centrality gravity*.

The edge betweenness centrality $c_B(e)$ corresponds to the number of shortest paths passing through a specific edge, formally defined as:

$$c_B(e) = \sum_{s,t \in V} \frac{\sigma(s, t|e)}{\sigma(s, t)} \tag{A.12}$$

Table A.1: Network topologies

Topology	$\|V\|$	$\|E\|$	$\|C\|$	$\|E\|/\|V\|$	$\|E\|/\|C\|$
GEANT	23	38	4	1.65	9.5
GARR	42	51	7	1.21	7.29
Uninett	66	91	3	1.38	30.33
WIDE	29	30	2	1.03	15
RedIris	19	31	5	1.63	6.2

where V is the set of nodes, $\sigma(s,t)$ is the number of shortest (s,t) paths, and $\sigma(s,t|e)$ is the number of those paths passing through edge e. In our problem, we assume that no information about link weights or routing tables is available. Therefore shortest paths are calculated using unitary link weight.

The degree centrality gravity of a link G_{C_D} corresponds to the product of the degree centralities (*i.e.,* the number of neighbours) of the link's endpoints u and v.

If there are unidirectional links in the network, this is calculated as:

$$G_{C_D}(u,v) = \text{outdeg}(u) \times \text{indeg}(v) \tag{A.13}$$

where u and v are, respectively, the egress and ingress nodes of the link. Otherwise, it is calculated as:

$$G_{C_D}(u,v) = \deg(u) \times \deg(v) \tag{A.14}$$

The communicability centrality gravity of a link corresponds to the product of the communicability centralities of the link's endpoint. The communicability centrality of a node [55], sometimes referred to as *subgraph centrality* corresponds to the number of distinct closed walks passing through that node. This metric captures how well connected is a node within a subgraph and is formally defined as:

$$G_{SC}(u,v) = \sum_{j=1}^{N} (v_j^u)^2 e^{\lambda_j} \times \sum_{j=1}^{N} (v_j^v)^2 e^{\lambda_j} \tag{A.15}$$

where v_j is an eigenvector of the adjacency matrix of G corresponding to the eigenvalue λ_j.

We implemented functions in FNSS to calculate these three metrics (edge betweeness centrality, degree centrality gravity and communicability centrality gravity) and assign link capacities so that the capacity of each link is proportional to the value of the selected metric associated to the link, as expressed in equation A.9.

A.3.2 Performance evaluation

We measure the performance of the proposed algorithms on five different backbone networks whose topology and link capacity assignments are known. These networks, also listed in Tab. A.1, are:

- **GEANT:** European academic network
- **WIDE:** Japanese academic network
- **GARR:** Italian academic network

- **RedIris:** Spanish academic network
- **Uninett:** Norwegian academic network

A first evidence of the existence of a correlation between capacities and centrality metrics is provided by the analysis of the Pearson's r between these two values. This evaluation, whose results are reported in Fig. A.4, shows that r is always positive and in most cases its value is greater than 0.3, which indicates a medium to strong correlation between the two variables.

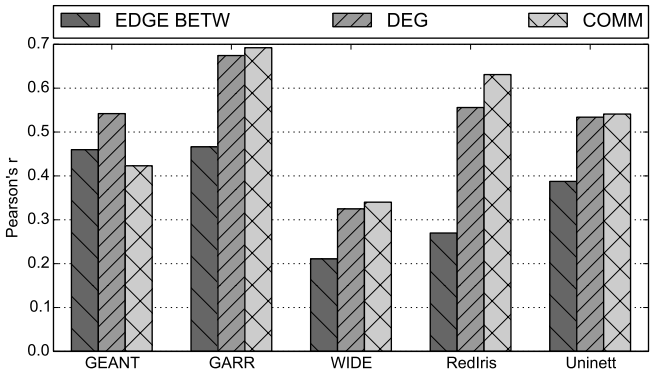


Figure A.4: Pearson's r between link capacity and centrality metrics

To further validate the performance of our methods, we assign link capacities according to the algorithms presented above and measure how close our estimation is from real capacity assignments. Such closeness is measured with two metrics.

The first metric used, which we called *Matching Capacity Ratio (MCR)*, measures the number of links whose estimated capacity corresponds exactly to the real capacity, normalised by the total number of links. This metric is formally defined as:

$$\text{MCR} = \frac{|\{e \in E|\, \hat{f}(e) = f(e)\}|}{|E|} \tag{A.16}$$

where E is the set of links, $f(e)$ and $\hat{f}(e)$ are, respectively, the real and assigned link capacity.

The second metric, which we called *Capacity Rank Error (CRE)*, captures the Root-Mean-Square Error (RMSE) between real and assigned capacities, and is defined as:

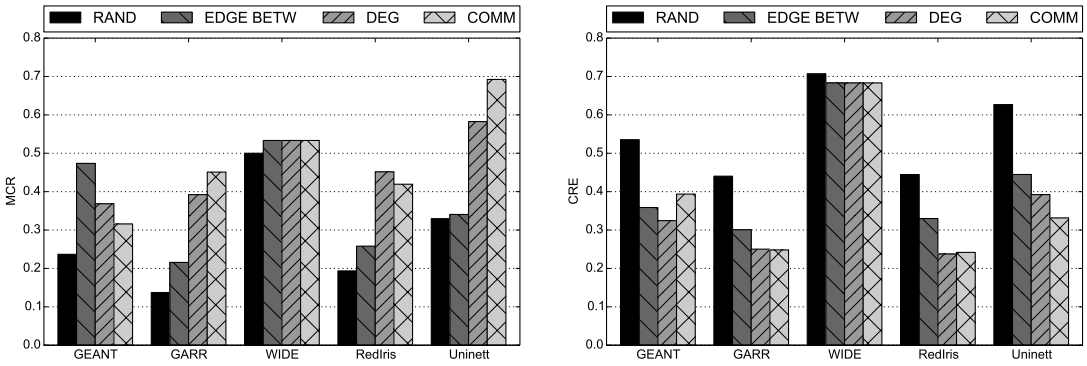
$$\text{CRE} = \frac{1}{|C|-1} \times \sqrt{\frac{1}{|E|} \sum_{e \in E} \left[R(\hat{f}_e) - R(f_e)\right]^2} \tag{A.17}$$

where C is the set of capacities and $R(x)$ is a function for values $x \in C$ such that:

$$R(x) = i \Leftrightarrow x = c_i \tag{A.18}$$

i.e., that returns the rank of a capacity value in the set of capacities C. For example, if the set of capacities C comprises 10, 40 and 100 Gbps and a link has capacity $c = 10$ Gbps, then $R(c) = 1$, if $c = 40$ Gbps, $R(c) = 2$ and if $c = 100$ Gbps, $R(c) = 3$. So, if on a link whose real capacity is 100 Gbps the algorithm assigns a capacity $\hat{c} = 40$ Gbps, then $R(\hat{c}) - R(c) = 1$, while if it assigns assigns a capacity $\hat{c} = 10$ Gbps, then $R(\hat{c}) - R(c) = 2$

The performance of our algorithms, with respect to the two metrics defined above, is shown in Fig. A.5, where they are compared with random assignments carried out using a uniform distribution. The RAND value represented in both figures refers to the median value of the metrics over 10^4 random assignments.



(a) Matching Capacity Ratio (MCR)

(b) Capacity Rank Error (CRE)

Figure A.5: Performance of link capacity assignment algorithms

These results show that in all cases considered, our algorithms yield better performance than random assignment, in terms of both MCR and CRE.

The smallest improvements are noticed in the WIDE topology. The reason justifying this result is that this topology comprises 30 edges, 16 of which having one capacity and 14 another one. Therefore, in this particular case, uniform random assignment, which assigns on average each capacity to 15 edges, can achieve reasonably good performance. In all other networks, link capacities are not distributed as uniformly as in WIDE, therefore the performance of uniform random assignments is lower. Anyway, it should be noted that even in the WIDE case, although the number of link with a given capacity fits very well a uniform distribution, our algorithms still perform better under both metrics considered.

Although further work is required to validate the effectiveness of these models, we think that these results are encouraging and that these models could be successfully adopted to improve the reliability of network simulations. In any case, further studies of these models will be part of our future work.

In conclusion, we are confident that our algorithms could be used by network researcher to assign more realistic link capacities to networks with known topologies but unknown capacities, like inferred topologies such as those of the RocketFuel dataset.

A.4 Example of utilisation

We report in this section a simple example that demonstrates how different features of the FNSS toolchain can be used to create a configured network topology and generate a traffic matrix. This specific example shows how to use the core Python library to parse a topology from the RocketFuel dataset, configure capacities, delays, weights and buffer sizes, generate a cyclostationary traffic matrix and save both topology and traffic matrix on XML files.

First, we import all functions of the FNSS core library:

```
from fnss import *
```

Then we parse a topology from the RocketFuel dataset:

```
topo = parse_rocketfuel_isp_map("file.cch")
```

At this point, we configure the parsed topology. We assign capacities of 1, 10 and 40 Gbps proportionally to edge betweenness centrality, weights proportionally to the inverse of the capacities, constant delays of 2ms and, finally, buffer sizes equal to the bandwidth-product delay.

```
C = [1, 10, 40]
set_capacities_edge_betweenness(topo, C, 'Gbps')
set_weights_inverse_capacity(topo)
set_delays_constant(topo, 2, 'ms')
set_buffer_sizes_bw_delay_prod(topo)
```

After fully configuring the network topology, we generate a cyclostationary traffic matrix with 7 periods of 24 samples.

```
tm = sin_cyclostationary_traffic_matrix(
       topo, mean=0.5, stddev=0.05,
       gamma=0.8, log_psi=-0.33, delta=0.2,
       n=24, periods=7, max_u=0.9
       )
```

Finally, we export topology and traffic matrix objects to XML files in order to be imported in the preferred target simulator.

```
write_topology(topo, 'topology.xml')
write_traffic_matrix(tm, 'traffic-matrix.xml')
```

A.5 Related work

Many tools have been presented in literature to address some of the issues related to the setup of network experiment scenarios. However, most of them only focus on the generation or parsing of topologies. Very little work has been done to support the entire setup process, being not tied to a single simulator or emulator.

The most common network simulators, such as ns-2 and ns-3 already provide support for parsing topologies from various sources, including RocketFuel and CAIDA datasets. However, these datasets merely consist of unconfigured network topologies. In fact, with the exception of a small subset of RocketFuel topologies which include estimations of link weights [123], all remaining network and traffic parameters are not known and, therefore, they need to be manually provisioned in the simulator.

The situation is similar for synthetically generated topologies. There are many freely-available tools capable of generating large scale topologies in accordance to the most common models. Such tools

include BRITE [127], Inet [181], GT-ITM [74] and aSHIIP [165]. However, also in this case, generated topologies cannot be used without further configuration. The only exception is BRITE which includes a method for assigning link capacities and delays. However, as explained in more detail in Sec. A.3, its capacity assignments are unrealistic and, anyway, the assignment of link weights, protocol stacks and traffic characteristics are not supported in any of these tools.

To the best of our knowledge, the only work attempting to move in a similar direction of FNSS is represented by the *NSF Frameworks for ns-3* project [149], although its scope is widely different. This project aims at building a framework for automating the network simulation and data collection in ns-3. Its objectives are only marginally overlapping with FNSS. In fact, while the *NSF Frameworks for ns-3* project aims at building a complete framework for repeating experiments with different parameters and collecting results, which is outside FNSS scope, it only targets the ns-3 simulator while FNSS aims at being a cross-platform tool, targeting also custom-built simulators. In addition, scenario generation code from *Frameworks for ns-3* is not available yet to the community, except for a BRITE parser for ns-3. Finally, *Frameworks for ns-3* does not include traffic matrices generation, as it does not address flow-level simulations.

A.6 Conclusion

This chapter provided two main contribution.

The first contribution is the design and implementation of the *Fast Network Simulation Setup (FNSS)* toolchain, a comprehensive library allowing network researchers and engineers to simplify the execution of the tasks required to generate a scenario for a controlled network experiment. This library allows users to generate topologies or import them from datasets, configure them with all required parameters and generate event schedules and traffic matrices.

The second contribution consists in novel methods for modelling the distribution of link capacities in a backbone ISP network. These methods assign link capacities proportionally to the values of a number of link centrality metrics. Although the design of these methods is simple, their accuracy, evaluated on a set of real network topologies, significantly outperformed the most commonly used models. Further refinements of these methods and a more comprehensive evaluation will be carried out as part of future work.

Appendix B

Icarus: a simulator for networked caching systems

B.1 Introduction

As extensively discussed in this thesis, the staggering increase in content delivery traffic has resulted in considerable research effort focusing on the design of networked caching systems. However, this did not result in suitable tools for evaluating the performance of such systems being made available to the community.

Early work on Web caches and CDNs produced only a small number of simulators, some of which are no longer available or maintained. In addition, most of them only support the evaluation of a single cache rather than a network of caches. The few simulators supporting the evaluation of networks of caches (*e.g.,* [158]) mainly target the evaluation of CDN coordination mechanisms and their impact on latency and throughput rather than focusing on overall caching performance.

More recently, the momentum gained by ICN has triggered efforts towards the development of new simulation software (*e.g.,* [2], [31], [39]) to assess the performance and viability of the new networking paradigm. All these simulators support caching since it is a fundamental building block of the most prominent ICN proposals. However they all present one or more of the following fundamental limitations which make them unsuitable tools when the objective is primarily the evaluation of caching performance.

- **Support for only a specific architecture.** Most ICN simulators available have been built to evaluate the performance of a single architectural approach and have not been designed to be extensible to support other architectures.

- **Poor scalability.** Caching nodes may converge very slowly to their steady state, depending on traffic patterns. For this reason, simulations may require from few hundred thousands to tens of millions of requests to yield reliable results. Most ICN simulators have been designed focusing on other aspects, such as protocol interoperability and congestion control, which can be generally evaluated with far smaller simulations. As a result, they have been implemented with a considerably lower level of abstraction than what required for caching, which makes them unable to run large scale simulations within a reasonable time frame.

- **Inability to run trace-driven simulations.** The use of trace-driven simulations is necessary to produce reliable caching results. In fact, synthetic stationary workloads generally fail to capture temporal and spatial locality patterns typical of real content distribution traffic. No ICN simulators currently support this feature.

As a consequence, most research papers on network caching present results generated either by running small scale experiments using publicly available simulators or by using custom built simulators which are not made available to the community. In the specific case of caching in the context of ICN, Tortelli *et al.* [167] estimated that 66% of recently published papers either used a custom-built simulator or did not provide any information about the simulator used to produce results. Clearly, the lack of publicly available simulators suitable for this purpose can strongly affect the reliability of research results.

To address all these issues, this chapter presents the design and implementation of *Icarus*, a simulator specifically designed for evaluating the performance of networked caching systems. *Icarus* implements all required features that are missing from other simulators. The source code of the simulator has been made available to the community [84] and, at the time of writing, it has already been used by several researchers to produce results presented in peer-reviewed publications.

In designing *Icarus*, we focused on satisfying two key non-functional requirements.

- **Scalability.** This requirement has been identified because, as mentioned earlier, reliable caching results may require experiments encompassing millions of simulated content requests. This is required to reach steady state in all caching nodes. As a result, *Icarus* has been implemented to run experiments of that size in a reasonable time frame. As shown more in detail in Sec. B.4, *Icarus* is capable of simulating several hundred thousand requests per minute per core on commodity hardware. As a result, it could easily complete simulations consisting of millions of requests in few minutes.

- **Extensibility.** This requirement is fundamental to ease the adoption of this simulator by the community. In fact, since caching research is currently very active, simulators' requirements are also changing very fast. As a result, by focusing on achieving extensibility, we expect this tool to serve the purpose of a larger user base. We also expect that making this tool publicly available will improve reliability and reproducibility of research results.

The remainder of this chapter is organised as follows. Sec. B.2 describes the implementation of the *Icarus* simulator. Sec. B.3 describes the modelling tools provided by *Icarus*. Sec. B.4 evaluates the performance of the *Icarus* simulator in terms of CPU and memory utilisation. Sec. B.5 provides an overview of the related work in the area. Finally, Sec. B.6 summarises the contribution of this chapter and presents the conclusions.

B.2 Implementation

B.2.1 Architecture and design

As already discussed, *Icarus* has been designed to achieve *extensibility* and *scalability*.

We addressed the extensibility requirement with a number of design decisions.

First of all, we selected Python as programming language. This provides several advantages. For example, its high-level syntax, which can be learned with a modest learning curve. In addition, the availability of scientific, simulation and network modelling libraries such as NumPy/SciPy [98], NetworkX [76] and FNSS (see Appendix A) further simplifies the implementation of new functionalities by its users.

Extensibility has also been achieved by adopting appropriate design patterns providing good modularity. For example, in implementing components most likely to be replaced and extended in the future (*e.g.,* cache replacement policies, routing and caching strategies and topology factories) we used a proxy design pattern together with a plug-in registration system. We illustrate how this system works by showing how a new replacement policy can be added to *Icarus* (see snippet below). In this case, a user simply needs to implement the new policy in a class extending the `Cache` class and overriding relevant methods to implement the desired behaviour. On top of the class implementation, the decorator `register_cache_policy` is invoked to register this implementation to the cache policy registry under the name `FOO_POLICY`. This registration makes the implementation discoverable and usable by the simulator.

```
@register_cache_policy('FOO_POLICY')
class FooCache(Cache):
    ...

    def get(self, k):
        ...

    def put(self, k):
        ...
```

At this point, to run experiments using this policy, the user only needs to specify it in the configuration file, as shown here below.

```
...
policies = ['LRU', 'LFU', 'FOO_POLICY']
...
```

The scalability requirement has been addressed mainly by selecting appropriate network abstractions that provide a good tradeoff between realism and simplicity. A key design decision is to simulate cache operations at a flow-level granularity, where the smallest event unit is the download of a content object, even when evaluating the performance of caches operating at a packet-level granularity.

The effectiveness of the flow-level abstraction for evaluating the performance of packet-level caches has been recently validated by Tortelli *et al.* [166]. They simulated a set of scenarios using Icarus, ndnSim [2] and ccnSim [39] and showed that all three simulators yield remarkably close caching

performance results despite both ndnSim and ccnSim operate at a packet-level granularity. In addition, for the scenarios considered, Icarus also executed experiments faster than the other two simulators benchmarked. These findings show that the assumptions made do not affect results reliability while effectively speeding up experiment execution.

In addition, scalability has also been addressed by using highly optimised implementations of complex routines provided by the SciPy, NumPy and NetworkX packages and by optimising the code through extensive profiling.

B.2.2 Workflow

Icarus implements the workflow depicted in Fig. B.1. Accordingly, the programme execution is carried out following these four sequential steps.

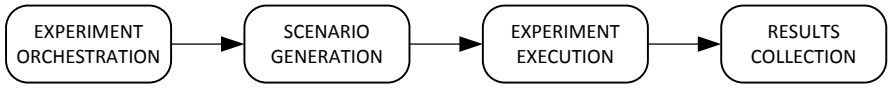


Figure B.1: Icarus workflow

- **Experiment orchestration.** This is the first stage of execution after the simulator is launched. The orchestration subsystem reads from a configuration file the scenario that the user desires to simulate (*e.g.,* topology, workload, strategy and cache replacement policy) as well as the metrics to be measured (*e.g.,* cache hit ratio and latency) and orchestrates the generation of a scenario, the execution of the experiment and the collection of results.

- **Scenario generation.** This stage comprises all the steps required to set up a fully configured network topology and a workload for the simulation. The scenario generation, as explained in Sec. B.2.4, heavily relies on functionalities offered by the FNSS toolchain presented in Appendix A.

- **Experiment execution.** This stage consists in the actual execution of the experiment. An instance of a simulation engine is provided with a scenario description (*i.e.,* a network topology and a workload). The engine reads events from the workload and dispatches them to the relevant handler. One or more data collectors measure various user-specified metrics and, at the end of the experiment, return results to the engine which will then pass them on to the results collection and analysis subsystem.

- **Results collection and analysis.** After each experiment terminates, results are collected in a result set which aggregates them and allows users to calculate confidence intervals, plot results or store data in various formats for subsequent processing.

In line with our objective to make *Icarus* extensible, all these four subsystems have been designed to be very loosely-coupled among each others. This has been achieved by using a façade design pattern to hide the internal implementation of each subsystem to other components and to use a simplified interface for interactions among them.

The following sections describe in detail how the stages presented above are implemented.

B.2.3 Experiment orchestration

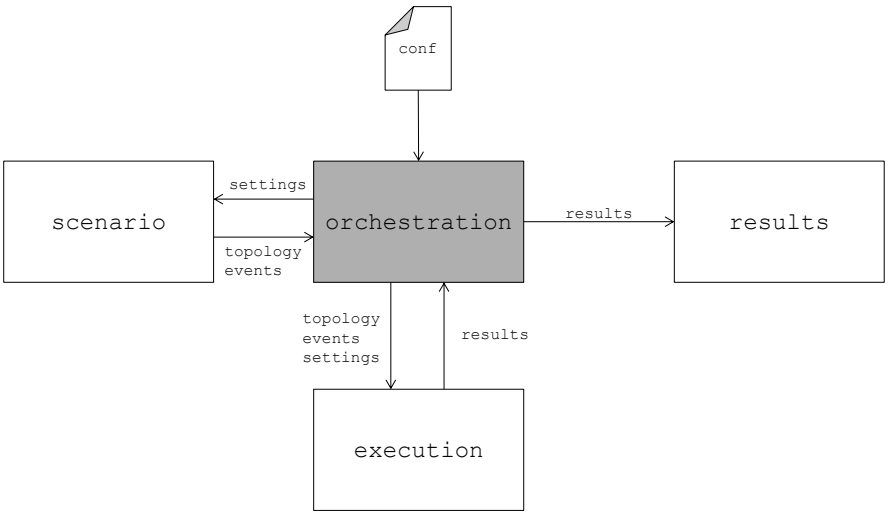


Figure B.2: Experiment orchestration

The experiment orchestration subsystem, depicted in Fig. B.2, is responsible for (i) parsing configuration parameters, (ii) request the generation of topologies and workloads to the scenario generation subsystem, (iii) dispatching them to the experiment execution subsystem and, finally (iv) collecting results and passing them on to the results collection subsystem.

The configuration is parsed from a Python-formatted file that is provided by the user as a command line parameter when *Icarus* is launched. The configuration file contains all information needed to execute the experiments. The configuration of each single experiment normally comprises topology, workload, content placement, cache placement, routing and caching strategy and performance metrics. The decision of formatting the configuration file as a Python source file simplifies the implementation of the parser. In addition, it allows users to use powerful Python features, such as list comprehension, to easily specify complex ranges and combinations of parameters.

After parsing the configuration, the orchestration subsystem validates the correctness of the input parameters. For each experiment instance, it first retrieves the relevant scenario from the scenario generation subsystem and then passes it to a process running an instance of the experiment execution subsystem.

Icarus supports the parallel execution of experiments to exploit the capabilities of multicore systems. However, it should be noted that it does not run a single experiment on multiple cores as in Parallel Discrete Event Simulations (PDES). Differently, *Icarus* allocates a single experiment to one core and parallel execution is achieved by running different experiments simultaneously on different cores.

The rationale behind this decision is that in caching research it is extremely rare to carry out performance evaluations with a single experiment. Instead, several experiments are executed using different combinations of parameters to evaluate results sensitivity against various factors or repeated several times to obtain statistically significant results. Therefore, as long as the configuration file requires to execute at least as many experiments as the number of available cores, all cores are fully utilised. We

show in Sec. B.4 that in a typical simulation campaign comprising several experiments, we can achieve a nearly linear speedup without PDES mechanisms.

After the experiment execution subsystem completes the simulation, it returns all results to the orchestrator, which simply passes them over to the results collection subsystem.

B.2.4 Scenario generation

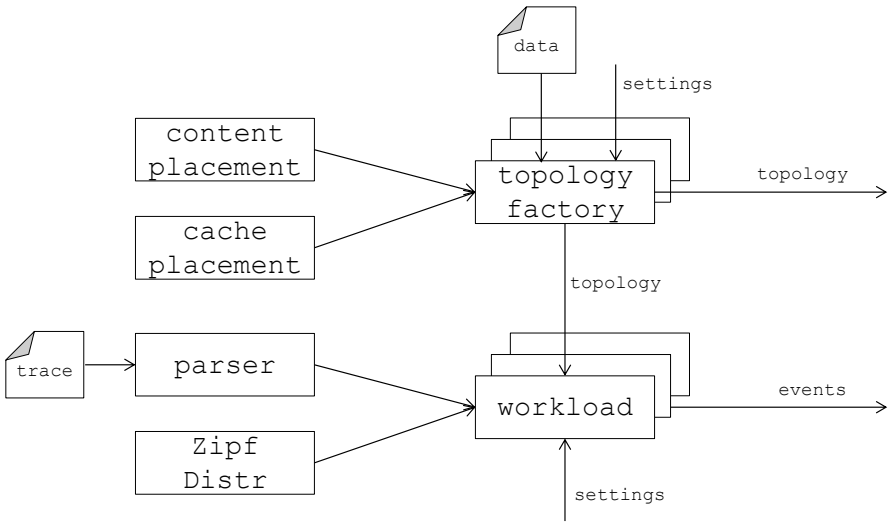


Figure B.3: Scenario generation

A simulation scenario is modelled as a tuple of two objects: a network topology and a schedule of events (see Fig. B.3).

A fully configured topology can be generated following three sequential steps. First, a topology factory function creates an FNSS `Topology` object whose edges are annotated with all required parameters (*e.g.,* link capacities, weights and delays) and whose nodes are annotated with their role (*i.e.,* receivers, origins or routers). The topology is also annotated with the list of cache candidate nodes, on which a cache may be deployed. Icarus already includes topology factory functions for creating topologies parsed from the RocketFuel dataset [157] as well as a some synthetic topologies. Second, a cache placement algorithm is applied to allocate caching space to (a subset of) cache candidate nodes. Third and finally, a content placement algorithm is applied to decide which content object should be permanently hosted by which origin node.

The event schedule is an iterator whose items are a pair containing:

- the timestamp at which the event occurs,
- a dictionary containing all the parameters describing the event, which normally comprises the node issuing a request, the content requested and a flag indicating whether the event must be logged.

This iterator is simply an interface provided to the simulation engine. The internal implementation can be easily modified or new implementations can be added without modifying any other component. Currently, *Icarus* provides functions to create request schedules in the following ways.

- Synthentically, with content requests generated following a Poisson distribution with Zipf-distributed content popularity.
- By parsing requests a traffic trace such as those provided by the IRCache dataset [87] or the Wikibench dataset [171].
- By parsing a complex synthetic workload generated by GlobeTraff [102].

B.2.5 Experiment execution

The experiment execution subsystem is responsible for running a single experiment and collecting results.

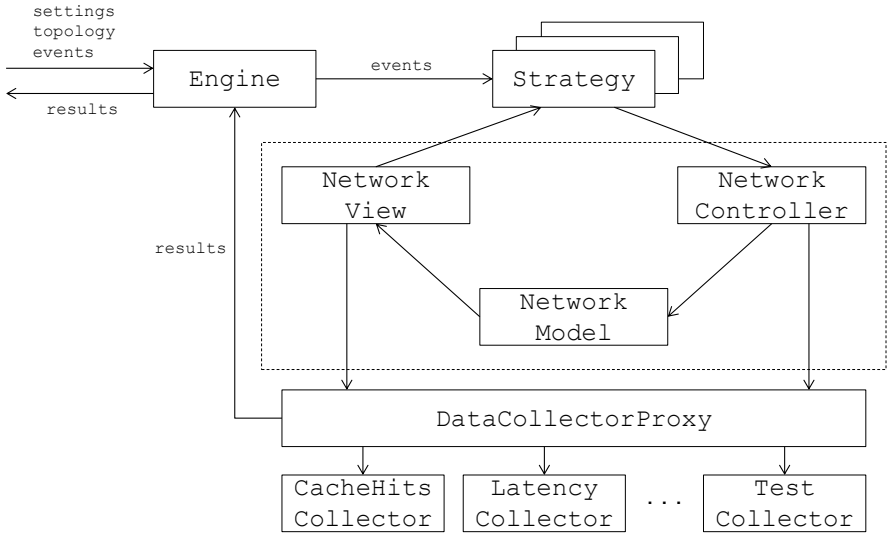


Figure B.4: Experiment execution flow diagram

As shown in Fig. B.4, this subsystem comprises the following four components:

- **`Engine`**: it is in charge of instantiating a caching and routing strategy for the experiment to which all events will be dispatched as they are read from a workload. Once an experiment is terminated, the engine is in charge of receiving results to pass to the results collection and analysis subsystem.
- **`Strategy`**: it implements the logic according to which requests and content items are routed and cached in the network. `Strategy` in reality is an abstract class, which can be extended by a concrete class implementing a caching and routing strategy. *Icarus* already provides implementations for the most common strategies (see sec. B.2.7). However, it is possible to easily implement a new strategy by overriding methods of the `Strategy` class without any modification to the rest of the code.
- **`Network`**: this component models the simulated network on which the `Strategy` component operates.

  It is implemented using a Model-View-Controller (MVC) design pattern. Accordingly, a strategy executes operations on the network exclusively by calling methods of the `NetworkController` object. The controller updates the state of the `NetworkModel` object, which maintains the current

state of the simulated network. In turn, the model updates the `NetworkView`, which provides the strategy with an up-to-date view of the network.

Implementing the network using an MVC design pattern is a departure from common network simulators, where the network is normally represented by an agent-based model in which each node is an agent. This design decision further increases the extensibility of the simulator because it strongly simplifies the implementation of a new strategy. In fact, this network abstraction makes it possible to implement new caching and routing strategies as centralised entities, without the need to be concerned about how to distribute and coordinate control logic across network nodes.

The `Network` also reports events to a data collector for gathering results. This design decision abstracts the complexity of data collection away from strategy implementation. This largely simplifies the implementation of new strategies, which we reckon will be a popular use case of our simulator, since research in request routing and content placement techniques is currently very active.

- **`DataCollector`**: it receives notifications about every event occurring in the network. The `DataCollector` component comprises a `DataCollectorProxy` and a number of data collector implementations each in charge of measuring a specific metric. The `DataCollectorProxy` interfaces directly with the `Network` component from which it receives all event notifications. Then, it dispatches events to various data collectors. The set of data collectors to which data is dispatched is specified by the user in the configuration file.

  Icarus provides collectors for measuring cache hit ratio, latency, link utilisation and path stretch. Similarly to the case of caching strategies, further data collectors can be implemented without any change to the rest of the code using a dedicated plug-in registration system.

  Once an experiment is terminated, the engine queries the data collector proxy for a summary of results. The latter gathers results from the various collectors and aggregates them in a dictionary, where each key is the identifier of the collector which gathered the data. Results are then passed back to the engine which passes them to the results collection and analysis subsystem. Finally, the analysis subsystem aggregates results from various experiments.

It should be noted that the experiment execution subsystem communicates with the experiment orchestration and results collection subsystem only via the `Engine` component. This is an implementation of the façade design pattern, where the `Engine` acts as an interface to other subsystems. This design decouples all other components of the experiment execution subsystem from the rest of the code.

B.2.6 Results collection and analysis

The main functionalities of the results collection and analysis subsystem, depicted in Fig. B.5, are to collect data from the various experiments, aggregate them and then process them. The aggregated results are stored in a specific data structure named `ResultSet`. This data structure is in practice a list of pairs albeit comprising additional methods for filtering and searching results. In each of these pairs, both

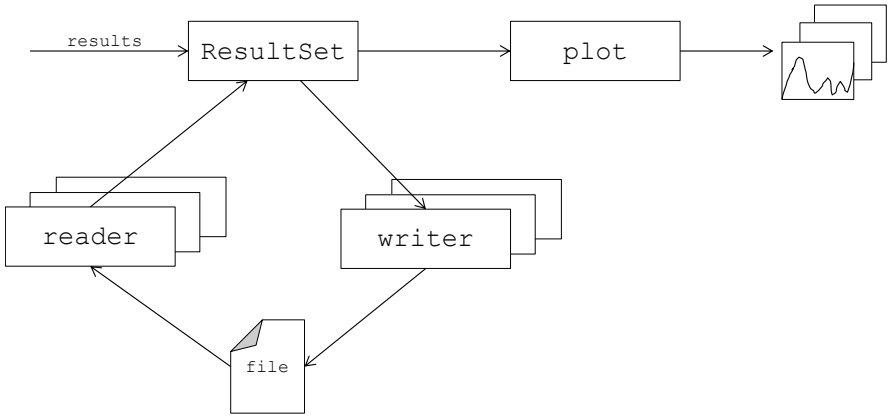


Figure B.5: Results collection and analysis

elements are dictionaries: the first stores all the parameters of the experiment and the second stores the results.

The result set can be stored on a file and later reloaded for further processing through specific `writer` and `reader` functions. *Icarus* currently supports only storing result sets as Python pickle files. However, additional reader and writer functions can be easily added to support other format, such as JSON or CSV.

Finally, this subsystem is responsible for analysing and plotting data. This subsystem can automatically calculate the confidence interval using the replications method and plot graphs with results of various metrics and various combinations of parameters. The automatic calculation of confidence intervals is in our opinion a fundamental feature contributing to the production of reliable results. This, in particular, is one of the objectives that motivated this work.

B.2.7 Caching and routing strategies

Icarus supports the following strategies:

- **Leave Copy Everywhere (LCE) [89]**: It is an on-path strategy replicating delivered content at each caching node on the delivery path.
- **Leave Copy Down (LCD) [113]**: It is an on-path strategy replicating delivered content only one level down the serving node.
- **Bernoulli random caching**: It is an on-path strategy randomly replicating content at each node traversed with probability p.
- **Random choice caching**: It is an on-path strategy replicating delivered content at only one randomly selected node along the delivery path.
- **Probabilistic Caching (ProbCache) [139]**: It is an on-path strategy replicating content randomly on the delivery path whereby the caching probability depends on a number of network parameters.
- **Cache Less for More (CL4M) [33]**: It is an on-path strategy replicating delivered contents only at the node of the delivery path with the greatest value of betweenness centrality.

- **Hash-routing**: It is a family of strategies routing requests to caches according to the value of a hash function computed on content item identifiers. Hash-routing strategies are discussed in greater detail in Chapter 4.

B.2.8 Cache replacement policies

Caching nodes can be configured to operate according to one of the following well-known cache replacement policies, which have all been presented in Chapter 2 and that we briefly recall here:

- **Least Recently Used (LRU)**: It replaces the least recently used item.
- **Perfect Least Frequently Used (Perfect-LFU)**: It replaces the least frequently used item measured keeping counters for all content items of the catalogue.
- **In-Cache Least Frequently Used (In-Cache-LFU)**: It replaces the least frequently used item, but differently from the Perfect-LFU case, it keeps counters only for content items in the cache.
- **First In First Out (FIFO)**: It replaces the item that was inserted in the cache first.
- **Random Replacement (RAND)**: It replaces a randomly selected item.
- **Segmented LRU (SLRU)**: It is a hierarchy of LRU caches in which an item is promoted to an upper segment when hit while in the segment below and demoted to a lower segment when evicted from the segment above. Items are always inserted at the top of the lowest segment and evicted from the cache when evicted from the lowest segment.
- **CLIMB**. It inserts items at the bottom of the queue and promote them one position at each hit. When a new item is inserted, the one at the bottom of the queue is replaced.

Tab. A.1 reports the computational complexity of the lookup and replacement operations for the policies listed above with respect to their *Icarus* implementation. The parameter C reported in the table refers to the size of the cache. As it can be observed from the table, all replacement policies can perform lookups in constant time. This owes to the fact that their implementation uses dictionaries for lookups. With respect to replacement, LRU, SLRU, FIFO, RAND and CLIMB all exhibit constant time complexity because their eviction queue is implemented using either a doubly-linked list (LRU, SLRU and CLIMB) or an array (FIFO and RAND). Differently, the two LFU policies have eviction queues implemented with a heap and therefore exhibit logarithmic time complexity. As shown in Sec. B.4, this greater complexity of the LFU cache results in slower execution of experiments in comparison to other replacement policies, especially when large caches are used.

Table B.1: Time complexity of search and replacement operations

Policy	Perfect-LFU	In-Cache-LFU	LRU	SLRU	FIFO	RAND	CLIMB
Lookup	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Replacement	$O(\log(C))$	$O(\log(C))$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$

B.3 Modelling tools

In addition to all the functionalities required for simulating networks of caches, *Icarus* also provides a set of modelling tools useful for caching research.

B.3.1 Modelling of caching performance

As discussed in Chapter 2, a considerable amount of work focused on modelling of caches, analysed both in isolation and as part of a network. *Icarus* includes implementations of some of these models, in order to cater for easy evaluation of their properties under different network conditions. We provide a brief summary of these models below.

B.3.1.1 Che's approximation

Icarus provides an implementation of the well know Che's approximation [35] to model the performance of an LRU cache subject to IRM demand.

The following snippet shows how to use this method to estimate the cache hit ratio of an LRU cache with 100 slots under stationary Zipf demand with coefficient $\alpha = 0.8$ and content catalogue of 1000 items.

```
>>> from icarus import *
>>> che_cache_hit_ratio(
          TruncatedZipfDist(alpha=0.8, n=1000).pdf,
          100)
0.36482948293429832
```

B.3.1.2 Laoutaris approximation

Icarus also provides the implementation of a method proposed by Laoutaris [112] to approximate the hit ratio of LRU caches under Zipf demand. This method provides a closed-form approximation of the cache hit ratio by modifying some steps of Che's approximation.

The following snippet shows how to use this method to estimate the cache hit ratio of an LRU cache with 100 slots under stationary Zipf demand with coefficient $\alpha = 0.8$ and content catalogue of 1000 items.

```
>>> from icarus import *
>>> laoutaris_cache_hit_ratio(0.8, 1000, 100)
0.35934820920359255
```

B.3.1.3 Numeric steady-state cache hit ratio

Finally, *Icarus* provides a function for estimating the steady-state cache hit ratio of a cache numerically.

Differently from the methods presented above, which can be used only with LRU caches, this method can be used with any cache implementation. In fact, it could be used with the cache replacement policies already provided or with a new one.

This method is useful for two purposes. First, to evaluate the cache hit ratio of a new cache replacement policy. Second, as a benchmark for a new cache hit ratio approximation of known cache replacement policies.

The following snippet shows how to use this method to calculate the cache hit ratio of an LRU cache with 100 slots under stationary Zipf demand with coefficient $\alpha = 0.8$ and content catalogue of 1000 items.

```
>>> from icarus import *
>>> numeric_cache_hit_ratio(
            TruncatedZipfDist(alpha=0.8, n=1000).pdf,
            LruCache(100))
0.37861264056574684
```

B.3.2 Analysis of content request traces

B.3.2.1 Estimation of content popularity skewness

Icarus provides a function to verify whether a given content request trace follows a Zipf distribution and to estimate what value of the Zipf coefficient (α) fits the trace best.

The estimation of the Zipf coefficient is carried out using the Maximum Likelihood Estimation (MLE) method. Once the α parameter is estimated, a χ^2 test is executed between the expected distribution with the estimated coefficient and the data.

The following snippet shows how to use this function to estimate the α parameter from a known Zipf distribution. The result is a tuple whose first item is the estimated α coefficient and second item is the p value that the given distribution actually follows a Zipf distribution with the estimated coefficient.

```
>>> from icarus import *
>>> zipf_fit(TruncatedZipfDist(0.8, 1000).pdf)
(0.79999999999571758, 1.0)
```

B.3.2.2 Trace parsers

Finally, *Icarus* provides a set of functions allowing users to parse content traces from the most common datasets. These traces can then be fed to a request generator and used in simulations or analysed to identify specific patterns.

Icarus supports parsing from two data formats. The first is the log format of Squid proxy. This is one of the most common open-source HTTP proxies available. This is also the format in which the traces provided by the IRCache project are made available [87]. The second is the Wikibench dataset [171]. This is a dataset of requests received by all Wikimedia websites over a period of time between 2007 and 2008.

B.4 Performance evaluation

In order to assess the scalability of the *Icarus* simulator and to validate its fitness for running large scale caching simulations, we evaluate its performance in terms of both CPU and memory utilisation under

varying conditions.

In particular, our analysis focuses on measuring the memory and processing footprint against various catalogue sizes and the speedup achieved when experiments are run in parallel on multiple cores.

B.4.1 Performance *vs.* catalogue size

The first set of experiments run, whose results are reported in Fig. B.6 and B.7, have been carried out to evaluate the sensitivity of execution time and memory utilisation against variations in sizes of caches and content catalogue. We report that we calculated the 95% confidence interval of those results using the replications method, but did not plot the related error bars because they were too small to be easily distinguishable from point markers.

These results have been gathered by running a set of simulations each consisting of 500,000 requests, generated following a Poisson distribution. Content popularity has been modelled as a Zipf distribution with coefficient $\alpha = 0.7$. The simulation scenario comprised a binary tree network topology with depth equal to 5 in which the root node was assigned as content source, the leaves as content receivers and all intermediate nodes as caches. In summary, such a topology comprises 63 nodes of which one is a source, 30 are caches and 32 are receivers. Routing and caching decisions have been taken according to the LCE strategy. Cache space has been assigned uniformly to all caching nodes. The cumulative size of caches in the network was equal to 10% of the content catalogue.

Experiments have been run with a representative sample of the replacement policies currently supported by *Icarus* (LRU, Perfect-LFU, FIFO and RAND) as well as with no caches (NULL). We evaluate performance in a range of content catalogue sizes from 10^3 to 10^7. It should be noted that since the ratio between cumulative cache size and content catalogue is fixed, a scenario with greater content catalogue also has greater cache sizes.

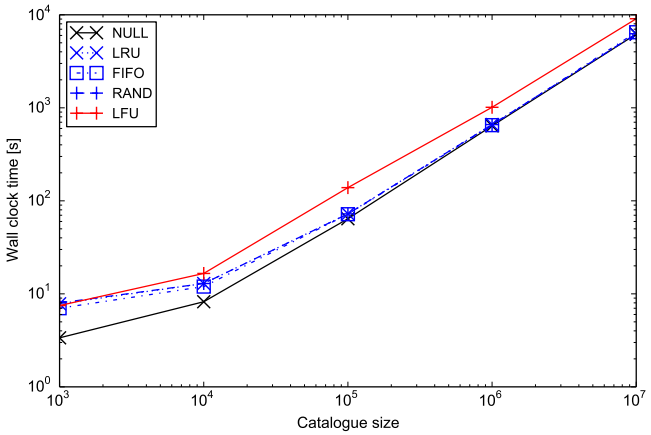


Figure B.6: Wall clock time *vs.* content catalogue size

Fig. B.6 shows the wall clock time elapsed between the beginning and the end of the experiment. As it can be observed from the graph, the execution time increases with the size of the content catalogue (and of the caches). This is caused mainly by the fact that in order to generate a random content identifier following a Zipf distribution, it is necessary to run a binary search in an array storing the cumulative

density function of the content popularity distribution. Although this routine has been implemented using the highly-optimised NumPy's `searchsorted` routine, such operation has still a $O(log(N))$ complexity (with N being the size of the content catalogue). The computational cost of these search tasks becomes considerably greater than the computational cost of LRU, FIFO and RAND caches operations for increasing content catalogues. This is evidenced by the fact that the execution time without caches (NULL) approaches those of LRU, FIFO and RAND policies when the catalogue size increases. In addition, while LRU, RAND and FIFO replacement policies yield comparable performance for all sizes of content catalogues investigated, the performance of Perfect-LFU policy degrades worse as the cache sizes increase. This is caused by the fact that while other replacement policies have a $O(1)$ cost for both search and replacement operations, the Perfect-LFU policy has a replacement cost of $O(C)$.

From the graph of Fig. B.6, it is also interesting noting the absolute time required to run these simulations, each consisting of 500,000 requests. For example, in the case of a content catalogue of 10^5 items and LRU replacement policy the mean execution time is nearly equal to 60 seconds, *i.e.*, one minute. These results, collected using dated hardware (CPU AMD Opteron 2347 with 1 GHz clock frequency), show that *Icarus* can actually scale very well and run realistic simulation scenarios with millions of requests in few minutes, even on modest hardware.

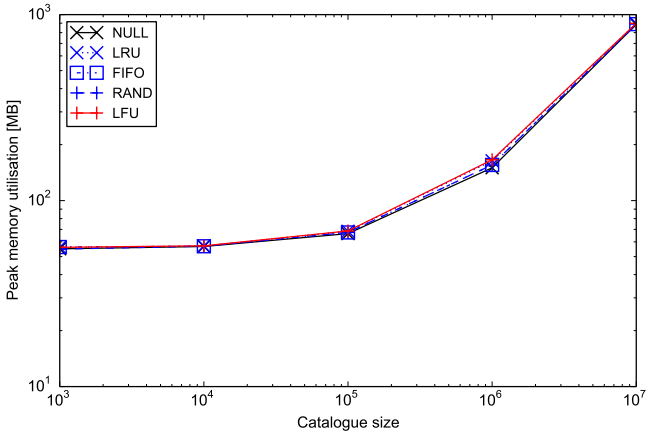


Figure B.7: Peak memory utilisation *vs.* content catalogue size

With respect to memory utilisation, whose results are depicted in Fig. B.7, we also observe a performance degradation as catalogue and cache sizes increase. This is caused by the memory occupied by the objects modelling caches and by the data structure mapping content identifiers to origin nodes. The latter is needed to route requests to a content origin. In this particular case, where the overall cache space accounts for 10% of the content catalogue, this data structure is responsible for a much greater memory consumption than cache objects. In fact, the line plot of memory consumption in the case of no caches (NULL) almost overlaps those representing cases where caches are used. It should be noted that the superlinear increase of memory utilisation is caused by the widespread use of dictionaries to implement caches and also the content-origin map. In fact, Python dictionaries have been adopted because they yield $O(1)$ lookup cost but do so at the cost of a superlinear memory utilisation.

Anyway, despite such superlinear increase in memory footprint, even in the largest scenario considered (*i.e.*, a catalogue of 10 million items and caches for 1 million items), the total memory utilisation is still below 1 GB. As a result, on most commodity machines currently available, it would be possible to run parallel experiments on all CPU cores without consuming all available memory. This memory footprint could be reduced and traded off with CPU utilisation by using a hash function to map content identifiers to source locations on-the-fly as opposed to using a precomputed hash table. However, since in the current implementation performance is normally CPU-bound, such alternative implementation would further increase the load on CPU resulting in performance degradation. Therefore, it is reasonable to maintain the current implementation.

B.4.2 Parallel execution speedup

We mentioned in Sec. B.2.3 that in the design of *Icarus* we decided not to employ any PDES mechanism to parallelise the execution of experiments. Instead we assumed that, since in caching research experiments are generally repeated with different parameters, we speculated that the parallel execution of separate experiments would yield satisfactory speedup performance in normal usage conditions. We also showed in Sec. B.4.1 that even with very large caches and content catalogues, the RAM required to run a single experiment is small enough to effectively enable the execution of parallel simulations on the same machine.

In this section we validate these assumption by running a combination of heterogeneous experiments using different number of cores and analyse the performance achieved in terms of speedup. We calculate the speedup as the ratio between the average wall clock time elapsed between the start of the first experiment and the end of the last experiment using a single core and the time required by the same workload using multiple cores.

The set of simulations executed consisted in 1280 distinct experiments, which have been generated from a combination of four different real topologies with variable sizes (parsed from publicly available datasets), four cache sizes, ten content popularity distributions and eight caching and routing strategies. Experiments have been designed to be as heterogeneous as possible and by selecting variable topology sizes and cache sizes in order to ensure that they would have different execution times. All experiments have been configured to have a content population of 10^7 object in order to increase memory footprint.

These experiments have been executed on a server equipped with two Quad-Core AMD Opteron processors (*i.e.,* 8 cores) and 32 GB of RAM and repeated using a variable number of cores from 1 to 8.

The results, depicted in Fig. B.8 show an almost linear speedup for the entire range considered. Moreover, even in the case of eight simultaneous experiments running, performance has always been CPU-bound and never reached the level required to consume all the available memory. In summary, this experiment validates the soundness of our design decision to implement parallel execution on a per-experiment granularity.

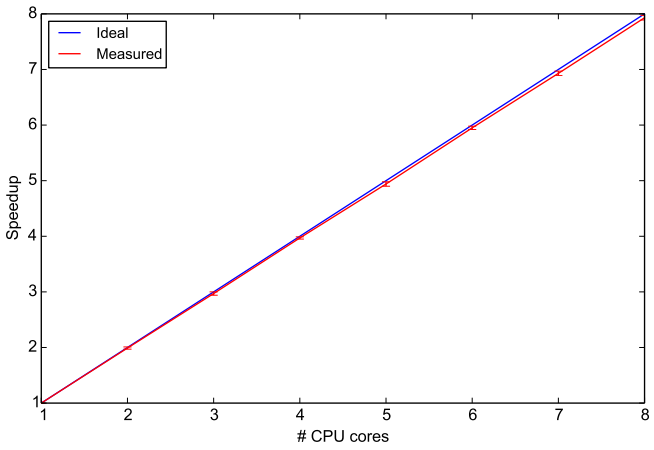


Figure B.8: Parallel execution speedup

B.5 Related work

A number of simulators supporting the evaluation of networks of caches have been developed in the context of either CDN or ICN research. However, none of them has been designed with the primary target of evaluating caching performance. As a consequence, all of them exhibit a number of limitations when specifically used for such objective.

In the context of CDNs, the main publicly available simulator is CDNsim [158], which is a packet-level simulator built on top of Omnet++ [174]. Its focus, however, is not on caching performance, but rather on the evaluation of latency and throughput taking into account the complete environment in which a CDN operates, *e.g.,* impact of DNS redirections and IP anycast. This is reflected in its design. In fact, its packet-level granularity strongly limits its scalability for evaluating caching performance.

In the context of ICN research, there are two main simulators supporting caching which are publicly available.

One is ndnSIM [2], which is built on top of ns-3 [77] and provides all the required functionalities to evaluate the transport behaviour of the NDN architecture [183]. Although ndnSim supports caching, it is not easily extensible to support architectures different from NDN and has a poor scalability, which makes it unsuitable for running large scale simulations needed for evaluating caching behaviour.

The other is ccnSim [39], which is a packet-level simulator based on Omnet++ [174] built mainly to evaluate the caching behaviour of the CCN architecture [89] and is the one closer to the work described in this chapter. Although ccnSim supports the simulation of large catalogues and cache sizes and it is shown to scale to simulations of millions of requests, its implementation is bound to the design of the CCN architecture. As a consequence, it is not suitable for evaluating the performance of caching systems in architectures different from CCN. For example, ccnSim does not allow asymmetric routing of request and content packets because not supported by the CCN architecture. In addition, it does not support the execution of trace-driven simulations, which is a feature of primary importance for caching research.

We argue that *Icarus* fills a gap in the simulation software for caching research, as it provides features necessary to accurately evaluate the performance of any networked caching system, without being tied to

only a specific architecture. Furthermore, its design allows for easy implementation of new features and strategies.

B.6 Conclusion

This chapter presented the design and implementation of *Icarus*, a scalable and extensible simulator for networks of caches. *Icarus* has been designed to address the shortcomings of currently available simulators, having in mind *extensibility* and *scalability* as main requirements.

The extensibility objective is of fundamental importance to facilitate its fast adoption by the research community. We addressed this requirement by appropriately using design patterns to effectively decouple the various components of the system.

The scalability objective is also fundamental for ensuring that the simulator is capable of running large scale simulations (which are required for producing statistically significant caching evaluations) in a reasonable time frame. We showed in our evaluation that *Icarus* scales well and is able to handle large content catalogues and caches with modest memory and CPU utilisation.

In conclusion, *Icarus* provides a set of utilities for modelling the performance of cache replacement policies and to analyse content request traces from widely used datasets. To the best of our knowledge, such functionalities are not provided by any other publicly available library.

Appendix C

Publications and patents

C.1 Publications

1. L. Saino, I. Psaras and G. Pavlou, Understanding sharded caching systems, in *Proceedings of the 35th IEEE International Conference on Computer Communications (INFOCOM'16)*, San Francisco, CA, USA, April 2016

2. R. Mansilha, L. Saino, M. Barcellos, M. Gallo, E. Leonardi, D. Perino and D. Rossi, Hierarchical content stores in high-speed ICN routers: Emulation and prototype implementation, in *Proceedings of the 2nd ACM conference on Information-Centric Networking (ICN'15)*, San Francisco, CA, USA, October 2015

3. I. Psaras, L. Saino and G. Pavlou, Revisiting resource pooling: the case for in-network resource sharing, in *Proceedings of the 13th ACM Workshop on Hot Topics in Networks (HotNets 2014)*, Los Angeles, CA, USA, October 2014

4. K. Katsaros, L. Saino, I. Psaras and G. Pavlou, On information exposure through named content, in *Proceedings of the International Workshop on Quality, Reliability, and Security in Information-Centric Networking (Q-ICN)*, Rhodes, Greece, August 2014

5. I. Psaras, L. Saino, M. Arumaithurai, K.K. Ramakrishnan and G. Pavlou, Name-based replication priorities in disaster cases, in *Proceedings of the 2nd IEEE workshop on Name Oriented Mobility (NOM'14)*, Toronto, Canada, April 2014

6. L. Saino, I. Psaras and G. Pavlou, Icarus: a caching simulator for Information Centric Networking (ICN), in *Proceedings of the 7th International ICST Conference on Simulation Tools and Techniques (SIMUTOOLS'14)*, Lisbon, Portugal, March 2014

7. L. Saino, I. Psaras and G. Pavlou, Hash-routing Schemes for Information Centric Networking, in *Proceedings of the 3rd ACM SIGCOMM Workshop on Information-Centric Networking (ICN'13)*, Hong Kong, China, August 2013

8. L. Saino, C. Corora and G. Pavlou, CCTCP: A scalable receiver-driven congestion control protocol for Content Centric Networking, in *Proceedings of the IEEE International Conference on*

Communications (ICC'13), Budapest, Hungary, June 2013

9. L. Saino, C. Cocora and G. Pavlou, A toolchain for simplifying network simulation setup, in *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques (SIMUTOOLS'13)*, Cannes, France, March 2013

C.2 Patent applications

1. M. Gallo, D.Perino and L. Saino, Method for managing a distributed cache, July 2015

Bibliography

[1] The Abilene topology and traffic matrices dataset. `http://www.cs.utexas.edu/~yzhang/research/AbileneTM/`. [Cited on pages 110, 113, and 116]

[2] A. Afanasyev, I. Moiseenko, and L. Zhang. ndnSIM: NDN simulator for NS-3. Technical Report NDN-0005, NDN, Oct. 2012. [Cited on pages 124, 126, and 139]

[3] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication (SIGCOMM'08)*, pages 63–74, New York, NY, USA, 2008. ACM. [Cited on pages 110 and 112]

[4] R. Albert and A. Barabási. Topology of evolving networks: local events and universality. *Physical review letters*, 85(24):5234–5237, 2000. [Cited on pages 106, 110, 111, and 117]

[5] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and Large CAMs for high performance data-intensive networked systems. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI'10)*, Berkeley, CA, USA, 2010. USENIX Association. [Cited on pages 79 and 100]

[6] A. Anand, V. Sekar, and A. Akella. SmartRE: An architecture for coordinated network-wide redundancy elimination. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication (SIGCOMM'09)*, pages 87–98, New York, NY, USA, 2009. ACM. [Cited on page 101]

[7] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*, pages 1–14, New York, NY, USA, 2009. ACM. [Cited on pages 50, 79, and 100]

[8] D. Applegate, A. Archer, V. Gopalakrishnan, S. Lee, and K. K. Ramakrishnan. Optimal content placement for a large-scale VoD system. In *Proceedings of the 6th International COnference on Emerging Network Experiments and Technologies (CoNEXT'10)*, pages 4:1–4:12, New York, NY, USA, 2010. ACM. [Cited on pages 20 and 21]

[9] A. Araldo, D. Rossi, and F. Martignon. Cost-aware caching: Caching more (costly items) for less (ISPs operational expenditures). *IEEE Transactions on Parallel and Distributed Systems*, PP(99), 2015. [Cited on page 77]

[10] M. Arumaithurai, J. Chen, E. Monticelli, X. Fu, and K. K. Ramakrishnan. Exploiting ICN for flexible management of Software-defined Networks. In *Proceedings of the 1st International Conference on Information-Centric Networking (ICN'14)*, pages 107–116, New York, NY, USA, 2014. ACM. [Cited on page 61]

[11] J. Ashayeri, R. Heuts, and B. Tammel. A modified simple heuristic for the p-median problem, with facilities design applications. *Robotics and Computer-Integrated Manufacturing*, 21(45):451 – 464, 2005. [Cited on pages 23 and 74]

[12] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'12)*, pages 53–64, New York, NY, USA, 2012. ACM. [Cited on pages 29 and 30]

[13] AT&T CDN Services. `http://goo.gl/KxmVAm`. [Cited on pages 17 and 58]

[14] O. I. Aven, E. G. Coffman, Jr., and Y. A. Kogan. *Stochastic Analysis of Computer Storage*. Kluwer Academic Publishers, Norwell, MA, USA, 1987. [Cited on page 25]

[15] A. Badam, K. Park, V. S. Pai, and L. L. Peterson. HashCache: Cache storage for the next billion. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*, pages 123–136, Berkeley, CA, USA, 2009. USENIX Association. [Cited on page 100]

[16] I. Baev, R. Rajaraman, and C. Swamy. Approximation algorithms for data placement problems. *SIAM J. Comput.*, 38(4):1411–1429, Aug. 2008. [Cited on page 20]

[17] S. Bansal and D. S. Modha. CAR: Clock with Adaptive Replacement. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST'04)*, pages 187–200, Berkeley, CA, USA, 2004. USENIX Association. [Cited on page 27]

[18] A. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999. [Cited on pages 106, 110, 111, and 117]

[19] T. Bektaş, J.-F. Cordeau, E. Erkut, and G. Laporte. Exact algorithms for the joint object placement and request routing problem in Content Distribution Networks. *Comput. Oper. Res.*, 35(12):3860–3884, Dec. 2008. [Cited on page 20]

[20] L. A. Bélády. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.*, 5(2):78–101, June 1966. [Cited on page 24]

[21] T. Benson, A. Anand, A. Akella, and M. Zhang. Understanding data center traffic characteristics. *SIGCOMM Comput. Commun. Rev.*, 40(1):92–99, Jan. 2010. [Cited on pages 110 and 112]

[22] D. S. Berger, P. Gland, S. Singla, and F. Ciucu. Exact analysis of TTL cache networks. *Performance Evaluation*, 79:2 – 23, 2014. [Cited on page 33]

[23] A. Bergman. Mysteries of a CDN explained. OmniTI Surge'12 conference, 2012. [Cited on pages 17 and 21]

[24] S. Borst, V. Gupta, and A. Walid. Distributed Caching Algorithms for Content Distribution Networks. In *Proceedings of the 29th IEEE International Conference on Computer Communications (INFOCOM'10)*, pages 1–9, March 2010. [Cited on page 20]

[25] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: evidence and implications. In *Proceedings of the 18th IEEE International Conference on Computer Communications (INFOCOM'99)*, volume 1, Mar 1999. [Cited on pages 24, 29, 30, and 79]

[26] T. Bu and D. Towsley. On distinguishing between Internet power law topology generators. In *Proceedings of the 21st IEEE International Conference on Computer Communications (INFOCOM'02)*, volume 2, pages 638 – 647 vol.2, 2002. [Cited on pages 106, 110, 111, and 117]

[27] The CAIDA AS relationships dataset. `http://www.caida.org/data/active/as-relationships/`. [Cited on pages 106 and 110]

[28] J. Cao, D. Davis, S. V. Wiel, and B. Yu. Time-varying network tomography: Router link data. *Journal of the American Statistical Association*, 95(452):pp. 1063–1075, 2000. [Cited on page 116]

[29] G. Carofiglio, M. Gallo, L. Muscariello, and D. Perino. Pending Interest Table sizing in Named Data Networking. In *Proceedings of the 2nd International Conference on Information-Centric Networking (ICN'15)*, pages 49–58, New York, NY, USA, 2015. ACM. [Cited on page 95]

[30] R. L. Carter and M. E. Crovella. Measuring bottleneck link speed in packet-switched networks. *Perform. Eval.*, 27-28:297–318, Oct. 1996. [Cited on page 117]

[31] Content-Centric Networking Packet Level Simulator. `http://systemx.enst.fr/ccnpl-sim`. [Cited on page 124]

[32] M. Cha, H. Kwak, P. Rodriguez, Y.-Y. Ahn, and S. Moon. Analyzing the video popularity characteristics of large-scale user generated content systems. *IEEE/ACM Transactions on Networking*, 17(5):1357–1370, Oct. 2009. [Cited on page 29]

[33] W. K. Chai, D. He, I. Psaras, and G. Pavlou. Cache less for more in information-centric networks (extended version). *Computer Communications*, 36(7):758 – 770, 2013. [Cited on pages 20, 21, 28, 70, and 132]

[34] W. K. Chai, N. Wang, I. Psaras, G. Pavlou, C. Wang, G. de Blas, F. Ramon-Salguero, L. Liang, S. Spirou, A. Beben, and E. Hadjioannou. CURLING: Content-ubiquitous resolution and delivery infrastructure for next-generation services. *IEEE Communications Magazine*, 49(3):112 –120, Mar. 2011. [Cited on page 18]

[35] H. Che, Y. Tung, and Z. Wang. Hierarchical Web caching systems: Modeling, design and experimental results. *IEEE Journal on Selected Areas of Communications*, 20(7):1305–1314, Sept. 2006. [Cited on pages 22, 23, 30, 32, 46, 68, and 134]

[36] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of Flash memory based solid state drives. In *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'09)*, pages 181–192, New York, NY, USA, 2009. ACM. [Cited on page 81]

[37] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture (HPCA'11)*, pages 266–277, Feb 2011. [Cited on page 81]

[38] F. Chen, R. K. Sitaraman, and M. Torres. End-user mapping: Next generation request routing for content delivery. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM'15)*, pages 167–181, New York, NY, USA, 2015. ACM. [Cited on pages 17 and 60]

[39] R. Chiocchetti, D. Rossi, and G. Rossini. ccnSim: an highly scalable CCN simulator. In *Proceedings of the 2013 IEEE International Conference on Communications (ICC'13)*, Budapest, Hungary, June 2013. [Cited on pages 124, 126, and 139]

[40] H.-g. Choi, J. Yoo, T. Chung, N. Choi, T. Kwon, and Y. Choi. CoRC: Coordinated Routing and Caching for Named Data Networking. In *Proceedings of the 10th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'14)*, pages 161–172, New York, NY, USA, 2014. ACM. [Cited on pages 34, 67, and 76]

[41] D. Cicalese, D. Joumblatt, D. Rossi, M. O. Buob, J. Auge, and T. Friedman. A fistful of pings: Accurate and lightweight anycast enumeration and geolocation. In *Proceedings of the 34th IEEE International Conference on Computer Communications (INFOCOM'15)*, pages 2776–2784, April 2015. [Cited on page 17]

[42] M. Claeys, D. Tuncer, J. Famaey, M. Charalambides, S. Latre, G. Pavlou, and F. De Turck. Proactive multi-tenant cache management for virtualized ISP networks. In *Proceedings of the 10th International Conference on Network and Service Management (CNSM'14)*, pages 82–90, Nov 2014. [Cited on page 21]

[43] C. Clos. A study of non-blocking switching networks. *Bell System Technical Journal*, 32(2):406–424, March 1953. [Cited on page 112]

[44] E. G. Coffman, Jr. and P. J. Denning. *Operating Systems Theory*. Prentice Hall Professional Technical Reference, 1973. [Cited on pages 29 and 35]

[45] F. J. Corbató. A paging experiment with the Multics system. Technical Report MAC-M-384, MIT, May 1968. [Cited on page 25]

[46] A. Dan and D. Towsley. An approximate analysis of the LRU and FIFO buffer replacement schemes. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'90)*, pages 143–152, New York, NY, USA, 1990. ACM. [Cited on pages 22 and 30]

[47] C. Dannewitz, D. Kutscher, B. Ohlman, S. Farrell, B. Ahlgren, and H. Karl. Network of Information (NetInf) - An Information-Centric Networking architecture. *Computer Communications*, 36(7):721 – 735, 2013. [Cited on page 18]

[48] B. Debnath, S. Sengupta, and J. Li. ChunkStash: Speeding up inline storage deduplication using Flash memory. In *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC'10)*, Berkeley, CA, USA, 2010. USENIX Association. [Cited on page 79]

[49] B. Debnath, S. Sengupta, and J. Li. FlashStore: High throughput persistent key-value store. *Proceedings of the VLDB Endowment*, 3(1-2):1414–1425, Sept. 2010. [Cited on pages 79 and 100]

[50] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM space skimpy key-value store on flash-based storage. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD'11)*, pages 25–36, New York, NY, USA, 2011. ACM. [Cited on pages 79 and 100]

[51] X. Dimitropoulos, D. Krioukov, M. Fomenkov, B. Huffaker, Y. Hyun, K. Claffy, and G. Riley. AS relationships: inference and validation. *SIGCOMM Comput. Commun. Rev.*, 37(1):29–40, Jan. 2007. [Cited on page 117]

[52] A. Downey. The structural cause of file size distributions. In *Proceedings of the 9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'01)*, pages 361–370, 2001. [Cited on page 30]

[53] EdgeCast, Inc. `http://www.edgecast.com/`. [Cited on pages 17 and 58]

[54] P. Erdős and A. Rényi. On the evolution of random graphs. *Magyar Tud. Akad. Mat. Kutató Int. Közl*, 5:17–61, 1960. [Cited on pages 106, 110, and 111]

[55] E. Estrada and J. Rodriguez-Velazquez. Subgraph centrality in complex networks. *Physical Review E*, 71(5):056103, 2005. [Cited on page 119]

[56] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication (SIGCOMM'99)*, pages 251–262, New York, NY, USA, 1999. ACM. [Cited on page 111]

[57] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC'11)*, pages 23:1–23:12, New York, NY, USA, 2011. ACM. [Cited on pages 41 and 63]

[58] S. K. Fayazbakhsh, Y. Lin, A. Tootoonchian, A. Ghodsi, T. Koponen, B. Maggs, K. Ng, V. Sekar, and S. Shenker. Less pain, most of the gain: Incrementally deployable ICN. In *Proceedings of the ACM SIGCOMM 2013 Conference on Data Communication (SIGCOMM'13)*, pages 147–158, New York, NY, USA, 2013. ACM. [Cited on page 29]

[59] P. Flajolet, D. Gardy, and L. Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Appl. Math.*, 39(3):207–229, Nov. 1992. [Cited on page 30]

[60] D. Floyer. Evolution of all-flash array architectures. `http://wikibon.org/wiki/v/Evolution_of_All-Flash_Array_Architectures`. [Cited on page 80]

[61] Fast Network Simulation Setup. `http://fnss.github.io/`. [Cited on page 107]

[62] N. Fofack, P. Nain, G. Neglia, and D. Towsley. Analysis of TTL-based cache networks. In *Proceedings of the 6th International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS'12)*, pages 1–10, ICST, Brussels, Belgium, Belgium, Oct 2012. ICST. [Cited on page 33]

[63] N. C. Fofack, M. Dehghan, D. Towsley, M. Badov, and D. L. Goeckel. On the performance of general cache networks. In *Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools (VALUETOOLS'14)*, pages 106–113, 2014. [Cited on page 33]

[64] N. Fotiou, D. Trossen, and G. Polyzos. Illustrating a publish-subscribe internet architecture. *Telecommunication Systems*, pages 1–13, 2011. [Cited on page 18]

[65] P. A. Franaszek and T. J. Wagner. Some distribution-free aspects of paging algorithm performance. *J. ACM*, 21(1):31–39, Jan. 1974. [Cited on page 24]

[66] C. Fricker, P. Robert, and J. Roberts. A versatile and accurate approximation for LRU cache performance. In *Proceedings of the 24th International Teletraffic Congress (ITC'12)*, pages 8:1–8:8, 2012. [Cited on pages 31 and 46]

[67] D. Fullagar. Designing Netflix's Content Delivery System. Uptime Institute Symposium, 2014. [Cited on pages 17 and 21]

[68] M. Gallo, B. Kauffmann, L. Muscariello, A. Simonian, and C. Tanguy. Performance evaluation of the random replacement policy for networks of caches. *Performance Evaluation*, 72:16 – 36, 2014. [Cited on page 32]

[69] M. Garetto, E. Leonardi, and S. Traverso. Efficient analysis of caching strategies under dynamic content popularity. In *Proceedings of the 34th IEEE International Conference on Computer Communications (INFOCOM'15)*, pages 2263–2271, April 2015. [Cited on page 29]

[70] N. Gast and B. Van Houdt. Transient and steady-state regime of a family of list-based cache replacement algorithms. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'15)*, pages 123–136, New York, NY, USA, 2015. ACM. [Cited on page 25]

[71] E. Gelenbe. A unified approach to the evaluation of a class of replacement algorithms. *IEEE Transactions on Computers*, C-22(6):611–618, June 1973. [Cited on pages 22, 25, and 30]

[72] Google Global Cache. `https://peering.google.com/about/ggc.html`. [Cited on pages 17, 21, and 22]

[73] R. Govindan and H. Tangmunarunkit. Heuristics for Internet map discovery. In *Proceedings of the 19th IEEE International Conference on Computer Communications (INFOCOM'00)*, volume 3, pages 1371–1380, 2000. [Cited on page 117]

[74] GT-ITM: Georgia Tech Internetwork Topology Models. `http://www.cc.gatech.edu/projects/gtitm/`. [Cited on page 123]

[75] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: a high performance, server-centric network architecture for modular data centers. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication (SIGCOMM'09)*, pages 63–74, New York, NY, USA, 2009. ACM. [Cited on pages 110 and 112]

[76] A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, Pasadena, CA USA, Aug. 2008. [Cited on pages 107 and 126]

[77] T. Henderson, S. Roy, S. Floyd, and G. Riley. ns-3 project goals. In *Proceedings of the 2006 workshop on ns-2: the IP network simulator*, page 13. ACM, 2006. [Cited on pages 107 and 139]

[78] J. H. Hester and D. S. Hirschberg. Self-organizing linear search. *ACM Comput. Surv.*, 17(3):295–311, Sept. 1985. [Cited on page 25]

[79] Y.-J. Hong and M. Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC'13)*, pages 13:1–13:17, New York, NY, USA, 2013. ACM. [Cited on page 39]

[80] S. Hosoki, S. Arakawa, and M. Murata. A model of link capacities in ISP's router-level topology. *Proceedings of the International Conference on Autonomic and Autonomous Systems*, 0:162–167, 2010. [Cited on page 117]

[81] HTTP archive. `http://httparchive.org`. [Cited on page 65]

[82] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of Facebook photo caching. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*, pages 167–181, New York, NY, USA, 2013. ACM. [Cited on pages 17 and 21]

[83] Q. Huang, H. Gudmundsdottir, Y. Vigfusson, D. A. Freedman, K. Birman, and R. van Renesse. Characterizing load imbalance in real-world networked caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks (HotNets-XIII)*, pages 8:1–8:7, New York, NY, USA, 2014. ACM. [Cited on pages 34, 36, and 39]

[84] Icarus caching simulator. `http://icarus-sim.github.io/`. [Cited on page 125]

[85] C. Imbrenda, L. Muscariello, and D. Rossi. Analyzing cacheable traffic in ISP access networks for micro CDN applications via content-centric networking. In *Proceedings of the 1st International Conference on Information-centric Networking (ICN'14)*, pages 57–66, New York, NY, USA, 2014. ACM. [Cited on page 29]

[86] Intel, Inc. DPDK framework. `http://dpdk.org`. [Cited on page 95]

[87] The IRCache project. `http://ircache.net/`. [Cited on pages 69, 130, and 135]

[88] V. Jacobson. Pathchar: A tool to infer characteristics of Internet paths, 1997. [Cited on page 117]

[89] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named content. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies (CoNEXT'09)*, pages 1–12, New York, NY, USA, 2009. ACM. [Cited on pages 13, 18, 21, 58, 78, 101, 132, and 139]

[90] P. Jelenković and A. Radovanović. Asymptotic insensitivity of Least-Recently-Used caching to statistical dependency. In *Proceedings of the 22nd IEEE International Conference on Computer Communications (INFOCOM'03)*, volume 1, pages 438–447, March 2003. [Cited on pages 29 and 31]

[91] P. R. Jelenković. Asymptotic approximation of the Move-to-Front search cost distribution and Least-Recently Used caching fault probabilities. *The Annals of Applied Probability*, 9(2):pp. 430–464, 1999. [Cited on page 31]

[92] P. R. Jelenković and X. Kang. Characterizing the miss sequence of the LRU cache. *SIGMETRICS Perform. Eval. Rev.*, 36(2):119–121, Aug. 2008. [Cited on pages 22, 32, 51, and 53]

[93] P. R. Jelenković and A. Radovanović. Least-Recently-Used caching with dependent requests. *Theoretical Computer Science*, 326:293 – 327, 2004. [Cited on page 31]

[94] P. R. Jelenković, A. Radovanović, and M. S. Squillante. Critical sizing of LRU caches with dependent requests. *Journal of Applied Probability*, 43(4):pp. 1013–1027, 2006. [Cited on page 31]

[95] jFed: Java based framework to support SFA testbed federation client tools. `http://jfed.iminds.be/`. [Cited on page 107]

[96] A. Jiang and J. Bruck. Optimal content placement for en-route Web caching. In *Proceedings of the 2nd IEEE International Symposium on Network Computing and Applications (NCA'03)*, pages 9–16, April 2003. [Cited on page 21]

[97] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*, pages 439–450, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc. [Cited on page 26]

[98] E. Jones, T. Oliphant, and P. Peterson. SciPy: Open source scientific tools for Python. `http://www.scipy.org/`. [Cited on page 126]

[99] R. Karedla, J. Love, and B. Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, March 1994. [Cited on page 26]

[100] 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. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC'97)*, pages 654–663, New York, NY, USA, 1997. ACM. [Cited on pages 34, 36, 58, 60, and 76]

[101] O. Kariv and S. L. Hakimi. An algorithmic approach to network location problems. ii: The p-medians. *SIAM Journal on Applied Mathematics*, 37(3):pp. 539–560, 1979. [Cited on page 23]

[102] K. V. Katsaros, G. Xylomenos, and G. C. Polyzos. GlobeTraff: A traffic workload generator for the performance evaluation of future internet architectures. In *Proceedings of the 5th IFIP International Conference on New Technologies, Mobility and Security (NTMS'12)*, pages 1–5, 2012. [Cited on page 130]

[103] L. Kaufman and P. J. Rousseeuw. *Partitioning Around Medoids (Program PAM)*, pages 68–125. John Wiley & Sons, 2008. [Cited on page 75]

[104] W. F. King III. Analysis of demand paging algorithm. In *Proceedings of the IFIPS congress*, 1971. [Cited on page 30]

[105] S. Knight, A. Jaboldinov, O. Maennel, I. Phillips, and M. Roughan. AutoNetkit: Simplifying large scale, open-source network experimentation. *SIGCOMM Comput. Commun. Rev.*, 42(4):97–98, Aug. 2012. [Cited on page 107]

[106] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765 –1775, Oct. 2011. [Cited on page 110]

[107] R. Kohavi and R. Longbotham. Online experiments: Lessons learned. *Computer*, 40(9):103–105, Sept 2007. [Cited on page 13]

[108] T. Koponen, M. Chawla, B.-G. Chun, A. Ermolinskiy, K. H. Kim, S. Shenker, and I. Stoica. A data-oriented (and beyond) network architecture. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM'07)*, pages 181–192. ACM, 2007. [Cited on page 18]

[109] P. Krishnan, D. Raz, and Y. Shavitt. The cache location problem. *IEEE/ACM Transactions on Networking*, 8(5):568–582, Oct. 2000. [Cited on page 23]

[110] D. Kutscher, S. Eum, K. Pentikousis, I. Psaras, D. Corujo, D. Saucez, T. Schmidt, and M. Waehlisch. ICN research challenges. *IRTF, draft-irtf-icnrg-challenges-02*, September 2015. [Cited on page 20]

[111] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: Rapid prototyping for Software-defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks (Hotnets-IX)*, pages 19:1–19:6, New York, NY, USA, 2010. ACM. [Cited on page 107]

[112] N. Laoutaris. A Closed-Form Method for LRU Replacement under Generalized Power-Law Demand. *ArXiv e-prints*, May 2007. [Cited on pages 31 and 134]

[113] N. Laoutaris, H. Che, and I. Stavrakakis. The LCD interconnection of LRU caches and its analysis. *Perform. Eval.*, 63(7), July 2006. [Cited on pages 20, 21, 23, 27, 32, 64, 70, and 132]

[114] N. Laoutaris, S. Syntila, and I. Stavrakakis. Meta algorithms for hierarchical Web caches. In *Proceedings of the IEEE International Conference on Performance, Computing, and Communications*, pages 445–452, 2004. [Cited on pages 27, 64, and 70]

[115] N. Laoutaris, V. Zissimopoulos, and I. Stavrakakis. Joint object placement and node dimensioning for Internet content distribution. *Inf. Process. Lett.*, 89(6):273–279, Mar. 2004. [Cited on page 22]

[116] N. Laoutaris, V. Zissimopoulos, and I. Stavrakakis. On the optimization of storage capacity allocation for content distribution. *Computer Networks*, 47(3):409 – 428, 2005. [Cited on page 22]

[117] Level3 Content Delivery Network. `http://goo.gl/p9AyhC`. [Cited on pages 17 and 58]

[118] J. Li, H. Wu, B. Liu, J. Lu, Y. Wang, X. Wang, Y. Zhang, and L. Dong. Popularity-driven coordinated caching in Named Data Networking. In *Proceedings of the 8th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'12)*, pages 15–26, New York, NY, USA, 2012. ACM. [Cited on page 21]

[119] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, pages 1–13, New York, NY, USA, 2011. ACM. [Cited on pages 79 and 100]

[120] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI'14)*, pages 429–444, Berkeley, CA, USA, 2014. USENIX Association. [Cited on page 34]

[121] Limeligth Networks. `http://www.limelight.com/`. [Cited on page 17]

[122] B. M. Maggs and R. K. Sitaraman. Algorithmic nuggets in content delivery. *SIGCOMM Comput. Commun. Rev.*, 45(3):52–66, July 2015. [Cited on pages 21, 25, 27, 34, 82, and 83]

[123] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. Inferring link weights using end-to-end measurements. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurement (IMW'02)*, pages 231–236, New York, NY, USA, 2002. ACM. [Cited on page 122]

[124] B. Marczak, N. Weaver, J. Dalek, R. Ensafi, D. Fifield, S. McKune, A. Rey, J. Scott-Railton, R. Deibert, and V. Paxson. An analysis of China's "great cannon". In *5th USENIX Workshop on Free and Open Communications on the Internet (FOCI'15)*, Washington, D.C., Aug. 2015. USENIX Association. [Cited on page 12]

[125] Markets and markets, Inc. Content Delivery Network (CDN) Market by Solutions (Web Performance Optimization, Media Delivery, Cloud Storage and Data Security, Transparent Caching, Transcoding and Digital Rights Management and Analytics and Monitoring) - Global Forecast to 2020. `http://www.marketsandmarkets.com/Market-Reports/content-delivery-networks-cdn-market-657.html`. [Cited on page 13]

[126] V. Martina, M. Garetto, and E. Leonardi. A unified approach to the performance analysis of caching systems. In *Proceedings of the 33rd IEEE International Conference on Computer Communications (INFOCOM'14)*, pages 2040–2048, April 2014. [Cited on pages 23, 26, 30, 31, 33, 47, 50, 51, 52, 55, and 68]

[127] A. Medina, A. Lakhina, I. Matta, and J. Byers. BRITE: An approach to universal topology generation. In *Proceedings of the 9th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'01)*, pages 346–353. IEEE, 2001. [Cited on pages 110 and 123]

[128] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association. [Cited on page 27]

[129] N. Megiddo and D. S. Modha. System and method for implementing an adaptive replacement cache policy. Patent 6996676, Dec 2004. [Cited on page 27]

[130] memcached - a distributed memory object caching system. `http://memcached.org/`. [Cited on pages 80, 100, and 105]

[131] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI'13)*, pages 385–398, Berkeley, CA, USA, 2013. USENIX Association. [Cited on page 39]

[132] The network simulator NS-2. `http://www.isi.edu/nsnam/ns/`. [Cited on page 107]

[133] A. Nucci, A. Sridharan, and N. Taft. The problem of synthetically generating IP traffic matrices: initial recommendations. *ACM SIGCOMM Computer Communication Review*, 35(3):19–32, 2005. [Cited on pages 115 and 116]

[134] E. Nygren, R. K. Sitaraman, and J. Sun. The Akamai network: A platform for high-performance internet applications. *SIGOPS Oper. Syst. Rev.*, 44(3):2–19, Aug. 2010. [Cited on pages 17 and 21]

[135] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD'93)*, pages 297–306, New York, NY, USA, 1993. ACM. [Cited on page 26]

[136] V. Pacifici and G. Dan. Content-peering dynamics of autonomous caches in a Content-centric Network. In *Proceedings of the 32nd IEEE International Conference on Computer Communications (INFOCOM'13)*, pages 1079–1087, April 2013. [Cited on page 77]

[137] D. Perino and M. Varvello. A reality check for Content Centric Networking. In *Proceedings of the 1st ACM SIGCOMM Workshop on Information-centric Networking (ICN'11)*, pages 44–49, New York, NY, USA, 2011. ACM. [Cited on page 101]

[138] D. Perino, M. Varvello, L. Linguaglossa, R. Laufer, and R. Boislaigue. Caesar: A content router for high-speed forwarding on content names. In *Proceedings of the 10th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'14)*, pages 137–148, New York, NY, USA, 2014. ACM. [Cited on pages 34, 83, 91, and 95]

[139] I. Psaras, W. Koong Chai, and G. Pavlou. In-network cache management and resource allocation for information-centric networks. *IEEE Transactions on Parallel and Distributed Systems*, 25(11):2920–2931, Nov 2014. [Cited on pages 20, 21, 28, 64, 70, and 132]

[140] L. Qiu, Y. R. Yang, Y. Zhang, and S. Shenker. On selfish routing in Internet-like environments. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications (SIGCOMM'03)*, pages 151–162, New York, NY, USA, 2003. ACM. [Cited on page 117]

[141] M. Raab and A. Steger. Balls into bins - a simple and tight analysis. In *Randomization and Approximation Techniques in Computer Science*, volume 1518 of *Lecture Notes in Computer Science*, pages 159–170. Springer Berlin Heidelberg, 1998. [Cited on page 36]

[142] E. Rosensweig and J. Kurose. Breadcrumbs: Efficient, best-effort content location in cache networks. In *Proceedings of the 28th IEEE International Conference on Computer Communications (INFOCOM'09)*, pages 2631–2635, April 2009. [Cited on page 22]

[143] E. Rosensweig, J. Kurose, and D. Towsley. Approximate models for general cache networks. In *Proceedings of the 29th IEEE International Conference on Computer Communications (INFOCOM'10)*, pages 1–9, March 2010. [Cited on pages 23 and 33]

[144] E. Rosensweig, D. Menasche, and J. Kurose. On the steady-state of cache networks. In *Proceedings of the 32nd IEEE International Conference on Computer Communications (INFOCOM'13)*, pages 863–871, April 2013. [Cited on pages 32 and 33]

[145] K. Ross. Hash routing for collections of shared Web caches. *IEEE Network*, 11(6):37–44, Nov 1997. [Cited on pages 34, 58, and 76]

[146] D. Rossi and G. Rossini. On sizing CCN content stores by exploiting topological information. In *Proceedings of the 1st IEEE Workshop on Emerging Design Choices in Name-Oriented Networking (NOMEN'12)*, pages 280–285, March 2012. [Cited on page 23]

[147] G. Rossini and D. Rossi. Coupling caching and forwarding: Benefits, analysis, and implementation. In *Proceedings of the 1st International Conference on Information-centric Networking (ICN'14)*, pages 127–136, New York, NY, USA, 2014. ACM. [Cited on pages 22 and 33]

[148] G. Rossini, D. Rossi, M. Garetto, and E. Leonardi. Multi-Terabyte and multi-Gbps information centric routers. In *Proceedings of the 33rd IEEE International Conference on Computer Communications (INFOCOM'14)*, pages 181–189, April 2014. [Cited on page 101]

[149] Project Summary: CI-ADDO-EN: Frameworks for ns-3. `http://www.eg.bucknell.edu/~perrone/research-docs/NSFProjectSummary.pdf`. [Cited on page 123]

[150] S. Saha, A. Lukyanenko, and A. Yla-Jaaski. Efficient cache availability management in information-centric networks. *Computer Networks*, 84:32 – 45, 2015. [Cited on page 76]

[151] Sandvine, Inc. 2015 Latin America and North America Report. `https://www.sandvine.com/pr/2015/5/28/sandvine-in-the-americas-netflix-google-facebook-the-internet.html`. [Cited on page 12]

[152] M. Z. Shafiq, A. X. Liu, and A. R. Khakpour. Revisiting caching in Content Delivery Networks. In *Proceedings of the 2014 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'14)*, pages 567–568, New York, NY, USA, 2014. ACM. [Cited on pages 25 and 82]

[153] A. Sharma, A. Venkataramani, and R. K. Sitaraman. Distributing content simplifies ISP traffic engineering. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'13)*, pages 229–242, New York, NY, USA, 2013. ACM. [Cited on pages 20 and 21]

[154] W. So, T. Chung, H. Yuan, D. Oran, and M. Stapp. Toward Terabyte-scale caching with SSD in a Named Data Networking router. In *Proceedings of the 10th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'12), Poster session*, 2014. [Cited on pages 91 and 101]

[155] W. So, A. Narayanan, and D. Oran. Named Data Networking on a router: Fast and DoS-resistant forwarding with hash tables. In *Proceedings of the 9th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS'13)*, pages 215–226, 2013. [Cited on page 101]

[156] A. Soule, A. Nucci, R. Cruz, E. Leonardi, and N. Taft. How to identify and estimate the largest traffic matrix elements in a dynamic environment. In *Proceedings of the joint international conference on Measurement and modeling of computer systems (SIGMETRICS'04/Performance'04)*, pages 73–84, New York, NY, USA, 2004. ACM. [Cited on page 115]

[157] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP topologies with Rocketfuel. In *Proceedings of the 2002 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM'02)*, pages 133–145, New York, NY, USA, 2002. ACM. [Cited on pages 69, 106, 110, 117, and 129]

[158] K. Stamos, G. Pallis, A. Vakali, D. Katsaros, A. Sidiropoulos, and Y. Manolopoulos. CDNsim: A simulation tool for Content Distribution Networks. *ACM Trans. Model. Comput. Simul.*, 20(2):10:1–10:40, May 2010. [Cited on pages 124 and 139]

[159] D. Starobinski and D. Tse. Probabilistic methods for Web caching. *Perform. Eval.*, 46(2-3):125–137, Oct. 2001. [Cited on page 29]

[160] Y. Sun, S. K. Fayaz, Y. Guo, V. Sekar, Y. Jin, M. A. Kaafar, and S. Uhlig. Trace-driven analysis of ICN caching algorithms on video-on-demand workloads. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies (CoNEXT'14)*, pages 363–376, New York, NY, USA, 2014. ACM. [Cited on page 29]

[161] X. Tang and S. T. Chanson. Adaptive hash routing for a cluster of client-side web proxies. *J. Parallel Distrib. Comput.*, 64(10):1168–1184, Oct. 2004. [Cited on page 76]

[162] M. B. Teitz and P. Bart. Heuristic methods for estimating the generalized vertex median of a weighted graph. *Operations Research*, 16(5):pp. 955–961, 1968. [Cited on pages 23 and 74]

[163] D. G. Thaler and C. V. Ravishankar. Using name-based mappings to increase hit rates. *IEEE/ACM Transactions on Networking*, 6(1):1–14, Feb. 1998. [Cited on pages 34, 36, 59, and 76]

[164] Y. Thomas, G. Xylomenos, C. Tsilopoulos, and G. C. Polyzos. Object-oriented Packet Caching for ICN. In *Proceedings of the 2nd International Conference on Information-Centric Networking (ICN'15)*, San Francisco, CA, USA, Oct. 2015. [Cited on page 101]

[165] J. Tomasik and M.-A. Weisser. aSHIIP: Autonomous generator of random Internet-like topologies with inter-domain hierarchy. In *Proceedings of the IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'10)*, pages 388–390. IEEE, 2010. [Cited on pages 110 and 123]

[166] M. Tortelli, D. Rossi, G. Boggia, and L. A. Grieco. Cross-comparison of ICN software tools. Technical report, Politecnico di Bari, August 2014. [Cited on page 126]

[167] M. Tortelli, D. Rossi, G. Boggia, and L. A. Grieco. Pedestrian crossing: The long and winding road toward fair cross-comparison of ICN quality. In *Proceedings of the International Workshop on Quality, Reliability, and Security in Information-Centric Networking (Q-ICN)*, Aug. 2014. [Cited on page 125]

[168] S. Traverso, M. Ahmed, M. Garetto, P. Giaccone, E. Leonardi, and S. Niccolini. Unravelling the impact of temporal and geographical locality in content caching systems. *IEEE Transactions on Multimedia*, 17(10):1839–1854, Oct 2015. [Cited on page 29]

[169] Twitter, Inc. `fatcache`. `https://github.com/twitter/fatcache`. [Cited on page 100]

[170] G. Tyson, Y. Elkhatib, N. Sastry, and S. Uhlig. Demystifying porn 2.0: A look into a major adult video streaming website. In *Proceedings of the 2013 Conference on Internet Measurement Conference (IMC'13)*, pages 417–426, New York, NY, USA, 2013. ACM. [Cited on page 21]

[171] G. Urdaneta, G. Pierre, and M. van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 53(11):1830–1845, July 2009. [Cited on pages 69, 82, 130, and 135]

[172] L. G. Valiant and G. J. Brebner. Universal schemes for parallel communication. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing (STOC'81)*, pages 263–277, New York, NY, USA, 1981. ACM. [Cited on pages 61 and 71]

[173] V. Valloppillil and K. W. Ross. Cache array routing protocol v1.0. *Internet Draft draft-vinod-carp-v1-03.txt*, February 1998. [Cited on page 76]

[174] A. Varga and R. Hornig. An overview of the OMNeT++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems (SIMUTools'08)*, pages 1–10, 2008. [Cited on pages 107 and 139]

[175] Velocix. `http://www.velocix.com/`. [Cited on page 17]

[176] C. Villamizar and C. Song. High performance TCP in ANSNET. *SIGCOMM Comput. Commun. Rev.*, 24(5):45–60, Oct. 1994. [Cited on page 114]

[177] Cisco Visual Networking Index (VNI). `http://www.cisco.com/c/en/us/solutions/service-provider/visual-networking-index-vni/index.html`. [Cited on pages 12 and 13]

[178] Y. Wang, Z. Li, G. Tyson, S. Uhlig, and G. Xie. Design and evaluation of the optimal cache allocation for content-centric networking. *IEEE Transactions on Computers*, PP(99), 2015. [Cited on page 23]

[179] B. Waxman. Routing of multipoint connections. *IEEE Journal on Selected Areas in Communications*, 6(9):1617–1622, 1988. [Cited on pages 106, 110, 111, and 117]

[180] P. Wendell and M. J. Freedman. Going viral: Flash crowds in an open CDN. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference (IMC'11)*, pages 549–558, New York, NY, USA, 2011. ACM. [Cited on page 12]

[181] J. Winick and S. Jamin. Inet-3.0: Internet topology generator. Technical report, University of Michigan, 2002. [Cited on pages 110 and 123]

[182] M. Yu, W. Jiang, H. Li, and I. Stoica. Tradeoffs in CDN designs for throughput oriented traffic. In *Proceedings of the 8th COnference on emerging Networking EXperiments and Technologies (CoNEXT'12)*, New York, NY, USA, 2012. ACM. [Cited on page 117]

[183] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, K. C. Claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang. Named Data Networking. *SIGCOMM Comput. Commun. Rev.*, 44(3):66–73, July 2014. [Cited on pages 13, 18, 19, 21, 58, 78, 101, and 139]

[184] D. Zhou, B. Fan, H. Lim, D. G. Andersen, M. Kaminsky, M. Mitzenmacher, R. Wang, and A. Singh. Scaling up clustered network appliances with ScaleBricks. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM'15)*, pages 241–254, New York, NY, USA, 2015. ACM. [Cited on page 34]

[185] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, high performance Ethernet forwarding with CuckooSwitch. In *Proceedings of the 9th ACM Conference on Emerging Networking Experiments and Technologies (CoNEXT'13)*, pages 97–108, New York, NY, USA, 2013. ACM. [Cited on page 83]