The Dissertation Committee for Arunkumar Venkataramani
certifies that this is the approved version of the following dissertation:

# Mechanisms and Algorithms for Large-Scale Replication Systems

Committee:

Michael D. Dahlin, Supervisor

Lorenzo Alvisi

Sally Floyd

Simon Lam

Charles Gregory Plaxton

Harrick Vin

# Mechanisms and Algorithms for Large-Scale Replication Systems

by

**Arunkumar Venkataramani, B.Tech., M.S.**

**Dissertation**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**Doctor of Philosophy**

**The University of Texas at Austin**

December 2004

# Acknowledgments

This dissertation has been possible thanks to the help and collaborative support of several colleagues and friends.

I'm extremely happy to have been advised by Mike Dahlin through the course of my PhD. I'm thankful to him for being a role-model advisor, for his prudent, yet patient, and constructive critique of my ideas, for sharing and boosting my enthusiasm even on ideas further from his core research interests, for employing his superhuman coding skills before paper deadlines that let us insert yet another experiment to make the paper inch closer to the unattainable ideal of perfection, for teaching me the ropes at every stage of my graduate career, for taking and sending me places, for leading by example at every instance, for investing more hope in me than I did myself, and for making my PhD a very pleasant experience overall.

I'm thankful to Lorenzo Alvisi for the innumerable and enjoyable discussions, both technical and otherwise, that we shared, for taking a deep and active interest in my graduate and professional career, and for his wonderful course on distributed computing which was one of the reasons that drew me to this field. I'm thankful to Greg Plaxton for nurturing the theoretical face of my research persona, for his insistence on beautiful and unambiguous proofs, and for the long and rewarding discussions we shared (the longest was more than 8 hours on phone with a few minutes off for nourishment and other bodily activities).

I'm thankful to Harrick Vin for his insightful comments, criticism, questions,

discussions, and support that helped shape the ideas in this dissertation at various stages, to Sally Floyd for reading through the dissertation, providing helpful comments and criticism, and engaging in discussions that helped refine my ideas, and to all of the above people and Simon Lam for serving on my dissertation committee.

This dissertation has immensely benefited from the contributions of colleagues I have closely collaborated with, namely, Ravi Kokku, Xiaozhou Li, Sadia Sharif, and Praveen Yalagandula. Ravi and Praveen implemented the NPS Web prefetching system. Mitul and Xiazhou contributed to many of the ideas in Chapter 7. Ravi Kokku and Amol Nayate happily volunteered as sounding boards for almost every idea I have toyed with. I'm thankful to Lei Gao, Jean-Philippe Martin, Amol Nayate, Jian Yin, and Jiandan Zheng for their collaboration on other works not included in this dissertation, to Jairam Mudigonda, Rama Kotla, Jasleen Kaur, Taylor Riche, Sugat Jain and other member of LASR for discussions and attendance at practice talks for conferences, to Rama Kotla, Vishv Jeet, Ravi Kokku, Prem Melville, Srujana Merugu, Joseph Modayil, Aniket Murarka, and Amol Nayate for collaborative, editorial, moral, and gastronomical support during paper deadlines, and to Neha Kumar for proof-reading the dissertation, stylistic suggestions, and more.

I'm thankful to Sara Strandtman, Katherine Utz, and Gloria Ramirez for their impeccable administrative support, and to Gloria, in particular, for patiently putting up with and readily suggesting escape strategies out of every administrative soup that I faithfully landed in.

ARUNKUMAR VENKATARAMANI

*The University of Texas at Austin*

*December 2004*

# Mechanisms and Algorithms for Large-Scale Replication Systems

Publication No. _____

Arunkumar Venkataramani, Ph.D.
The University of Texas at Austin, 2004

Supervisor: Michael D. Dahlin

Replication of data and services is a fundamental building block in the design of distributed systems. Though replication has several benefits for large-scale systems distributed across wide-area networks, it also introduces considerable complexity over a centralized unreplicated system. This thesis contributes mechanisms and algorithms that enable the design of simpler and more efficient large-scale replicated systems.

On the mechanism front, we present aggressive speculative replication (ASR) as a basic primitive in building large-scale replicated systems. ASR refers to aggressively trading-off costs of hardware resources for making replicas of content and updates in a speculative manner in return for savings in human time and improved service quality. Today, replicated systems are unable to avail of the benefits of ASR as current architectures rely on manually-tuned parameters to manage the complexity of large-scale replicated systems. Consequently, such systems are hard to build and maintain, inefficient in utilizing available resources, and prone to the risk of overload. As a solution, we present an architecture, Mars, that performs ASR in

a self-tuning manner, i.e. without the need for manual tuning. To enable realization of Mars' architecture in practical systems, we build TCP Nice, an end-to-end transport protocol for background transfers. We demonstrate the benefits of Mars through a case study of a Web prefetching system, NPS, and show that the Mars approach simplifies the design, efficiently uses available resources to give performance benefits, is robust to the risk of system overload, and is easily deployable without changing existing infrastructure by using only simple end-to-end mechanisms.

On the algorithmic front, we make three contributions. First, we present a speculative replication algorithm, namely, long-term prefetching, to minimize average response times at a large cache constrained by bandwidth and validate its effectiveness through simulations using Web traces. Next, we develop a speculative replication algorithm for minimizing response times in a set of hierarchically-organized distributed cooperating caches constrained by bandwidth, and show that it is constant-competitive. Finally, we study the theoretically intriguing problem of minimizing average response times in a set of hierarchically-organized distributed cooperating caches constrained by storage space and show a nonconstant lower bound on the competitive ratio of any online algorithm that is allowed at most a constant-factor space advantage.

# Contents

# Chapter 1

# Introduction

This dissertation explores how large-scale replicated systems should be designed. *Large-scale replicated systems* are systems distributed across wide-area networks that involve movement of massive amounts of information.

Replication, or making copies, of data and services is a fundamental building block in the design of distributed systems. The idea of making a copy closer to the point of access for improving performance or reliability has been used in computing systems for several decades. Processors employ caching and prefetching techniques in memory hierarchies to mask long disk latencies. Software systems such as databases, network file systems, and replicated state machines use similar techniques for improving performance and reliability. Over wide-area networks (WANs), due to tremendous growth of the Internet in the last decade, replication is manifest in several commonly used distributed applications. Examples of such applications include browser and Web proxy systems that incorporate caching and prefetching, content distribution networks, replicated server systems, distributed databases, backup or mirrored systems, file sharing applications, edge-service architectures, web crawlers, automatic software updates etc.

## 1.1 Wide-area Network Replication

WANs are characterized by inherently long latencies and susceptibility to network partitions that render parts of the system inaccessible. In such an environment, replicating content has several attractive benefits that include (i) reduced response times due to proximity of the replica to the point of access, (ii) greater availability due to increased tolerance to network partitions – a suitably designed replicated system may continue to operate correctly despite the inaccessibility of a subset of replicas, (iii) support for ubiquitous access to data and operation in a disconnected mode for mobile users, (iv) improved throughput and service quality through selection of a good replica, (v) improved load balancing and provisioning of available resources in tune with the geographic distribution of demand, and (vi) increased tolerance to faults and malicious attempts to disrupt normal operation of the system.

However, replication across WANs introduces considerable complexity over a centralized unreplicated system. On one hand, making copies of objects raises policy questions such as what objects to replicate, where to place replicas, how to locate good replicas to service requests, and how to optimally replicate over a wide-area network constrained by bandwidth, or replica locations constrained by space, computing power and other resource constraints. On the other hand, replication also introduces mechanism issues such as how to tolerate network partitions and ensure graceful degradation of performance, how to satisfy application-specific consistency requirements of replicated content, and how to allocate resources to different forms of replication at different points in the network. Thus, replication is a complex multi-dimensional problem. Replication system designers seek to optimize metrics such as availability, response time, cost of replication, (actual) consistency of replicated data, while respecting constraints such as network connectivity, network bandwidth, storage space at replicas, computing constraints at replicas, and

(minimum) consistency requirements imposed by the application. We refer to this optimization problem as the general replication problem.

It is inherently difficult to balance the constraints against the metrics. For example, the CAP dilemma, suggested by Brewer [29] and subsequently formalized by Gilbert and Lynch [77], states that replicated systems cannot achieve both high **C**onsistency and high **A**vailability in a network prone to **P**artitions. Practical systems attempt to circumvent this reality by offering relaxed consistency guarantees such as eventual consistency [168, 108], or sacrificing availability when partitioned and performance when connected. Even for a fixed consistency requirement and network fault pattern, increasing the number of replicas beyond a point can hurt availability in certain scenarios [188]. In some implementations of transactional systems, increasing the scale of replication can lead to considerable human confusion – a ten-fold increase in the number of replicas can cause a thousand-fold increase in the number of reconciliations or deadlocks [81]– and possibly system delusion, rendering the database inconsistent with no obvious way of repair. It is of little wonder that few massive-scale replication systems over WANs offering strong consistency exist today .

One technique that can alleviate long latencies and network partitions in WANs is that of *speculative replication*, i.e. moving content to a location before it is accessed there. Speculative replication, also known as *prefetching* in the context of processor caches, database applications, Web content etc., is widely used to improve the performance of distributed applications. Examples include content distribution services that replicate content closer to clients before they request it, websites and browsers that prefetch hyperlinks or related webpages to improve response times perceived by clients, Bayou's [145] anti-entropy protocol for proactively exchanging updates, update propagation in Coda [108], Pangaea [158] and other replicated file systems, media content players downloading a minimum length of a file before

beginning to play them, proactive reconciliation mechanisms in applications like Lotus Notes and so on .

## 1.2 Aggressive Speculative Replication (ASR)

In this dissertation, we propose *aggressive speculative replication* as a fundamental design primitive for constructing large-scale replicated systems. Aggressive speculative replication, or *ASR*, implies that in periods of network connectivity, replicas are created speculatively on a massive scale at locations in the system where there is a non-zero probability of accessing the object, that updates to objects are propagated aggressively to all replica locations, and that the system strives to continually maintain all replicas as close to strong consistency as allowed by available hardware resources. Relating ASR to the metrics that the general replication problem seeks to optimize, we find that ASR improves availability because replica locations can mask network partitions for a longer duration if a sufficiently fresh copy is locally available, ASR improves response times as the requested content is fetched from a replica nearby. ASR improves consistency as the system aggressively propagates updates to replica locations.

Utilizing hardware resources to their fullest in order to perform ASR increases the cost of replication. However, this increase in cost is justified as we observe the following technology trends and characteristics of application workloads:

1. *Technology trends*: Hardware capacities – network bandwidth, computing power, and storage space – are increasing exponentially, while the corresponding costs are steadily falling. Optical channel capacities are increasing by around 100% a year [48]. Processing speeds are increasing by around 50% every year [3]. The cost per megabyte of disk drives has been halving approximately every nine months [153].

2. *Value of human time*: The value of human time is non-decreasing. Increasing capacities and decreasing costs of hardware resources make it more attractive to trade-off such costs in return for savings in human time and enhancing productivity. Such a trade-off is also advocated by Gray and Shenoy [82], and Chandra [37] who use a simple economic model that compares the costs and benefits of replication by converting them to the corresponding dollar values per unit.

3. *Workload characteristics*: Large-scale distributed systems often experience bursts in demand when the load on the system is significantly greater than the long-term average. For instance, on September 11, 2001, CNN.com experienced a load that exceeded the expected peak [119] by a factor of 20. On the same day, Akamai's servers experienced a load several 100 times the average [154]. The Zipfian distribution of popularity of Web objects suggests that as systems scale to larger communities, the ratio of peak to average load on a server is likely to increase further. Thus, the nature of these applications dictates that systems will be designed to be considerably over-provisioned in anticipation of load surges, and consequently have abundant spare capacity during periods of normal load. This spare capacity can be utilized for ASR in order to improve metrics like performance, availability, and consistency. Conversely, if such systems are designed to incorporate ASR mechanisms, they will be more tolerant to unusual peaks in load. Over-provisioning of systems is also in keeping with the above argument of "wasting" hardware resources for human time and productivity. A historic example of such a trend is the move from mainframe computers that use resources efficiently to personal computers that are inefficient but convenient.

Technology trends, application workloads, and human needs provide a strong case for incorporating ASR into large-scale replicated systems. However, current

operating systems provide little support for ASR, leaving application designers to grapple with the following hard challenges:

1. *Interference*: Interference refers to competition for resources between speculative and regular load in the system. This could have a net effect of degrading system performance. Interference is also undesirable due to its inherent unfairness – speculative load meant to improve one user's performance could interfere with regular load initiated by another user causing the latter to observe long response times. Having to account for interference makes design policies for speculative replication considerably complex in a large-scale distributed system.

2. *Utilization*: A goal conflicting with minimizing interference is that of ensuring utilization of available system resources for ASR. An ASR mechanism that is overly conservative may fail to give any performance or availability benefits, while an overly aggressive one may cause interference.

3. *Robustness*: Networks, web servers, and operating systems fundamentally behave like queueing systems where the response times assume low values when the system is lightly loaded and sharply start to rise as the load approaches system capacity. If not done carefully, ASR entails the risk of driving the load on the system, or parts thereof, beyond capacity and causing severe performance degradation to the extent of rendering a service unavailable.

4. *Self-tuning Support*: Due to inadequate system support for speculative replication, many application designers attempt to balance the above concerns by resorting to *manual-tuning*. In this approach, system parameters are hand-tuned to allocate resources for speculative and regular load in a non-interfering manner. For instance, some Web prefetching systems have been known to use static or workload-specific threshold values to limit the bandwidth consumed

by prefetched traffic [64, 140]. A core tenet of this dissertation is that such *threshold-based* approaches to tackle the problem of resource management in large-scale replicated systems are fundamentally flawed. It is the lack of adequate system support for ASR that forces application developers to employ short cuts that are complex in the long run, utilize resources inefficiently, and expose the system to the risk of overload. Hence, a pressing need for large-scale replicated systems today is *self-tuning* system support - mechanisms that render human intervention unnecessary - for aggressive speculative replication.

## 1.3 Contributions

This dissertation introduces and evaluates, by means of prototype experiments as well as analytical modeling, mechanisms and algorithms for large-scale replicated systems.

On the mechanism front, the main contributions of this dissertation are as follows:

1. We make a case for aggressive speculative replication as a fundamental design primitive in building large-scale replicated systems.

2. We demonstrate through prototype-based experiments how approaches relying on manually-tuned thresholds are fundamentally flawed due to their complexity, inefficiency, and the risk of system overload. We make a case for self-tuning system support for building large-scale distributed applications that involve massive replication.

3. We develop Mars, an architecture that provides self-tuning system support for performing ASR in a manner that prevents interference between speculative and regular load. We demonstrate that the Mars approach simplifies application design by not relying on manually-tuned thresholds, more efficiently

utilizes resources for improving performance and availability, and is safe from the risk of overloading system components.

4. We build usable prototypes to instantiate Mars' architecture in real-world replicated systems. In particular, to prevent network interference, we develop the abstraction of a *background transfer* that does not interfere with existing regular traffic in the network. Our network transport protocol, TCP Nice, provides this abstraction in a deployable manner with the modification of the sender-side congestion control protocol only. In addition, for greater deployablity, we develop a user-level implementation of TCP Nice to obviate a kernel level installation. TCP Nice effectively provides a two-level network-wide prioritization without making changes to any routers, and is the first such protocol to the best of our knowledge. TCP-LP [115], a similar protocol for low-priority transfers, was developed independently and almost simultaneously. The initial design, implementation, and technical results associated with TCP Nice previously appeared in the paper "TCP Nice: A Mechanism for Background Transfers", A. Venkataramani, R. Kokku, M. Dahlin published in "Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI '02)" and were presented at Boston, MA, USA in December 2002.

5. We demonstrate the benefits of Mars' architecture through a case study of NPS, a prototype Web prefetching system that is non-interfering and easily deployable. NPS is free from the vagaries of technology trends, workloads, and estimation error that static thresholds are subject to, and can still provide significant reductions in response times commensurate with available spare capacity. The initial design, implementation, and technical results associated with NPS previously appeared in the paper "NPS: A Non-Interfering Deployable Web Prefetching System" published in "Proceedings of the Fourth

USENIX Symposium on Internet Technologies and Systems (USITS '03)" and were presented at Seattle, WA, USA in March 2003.

The goal of the algorithmic half of the dissertation is to augment the mechanism contributions with appropriate policies for selection and placement of speculatively replicated objects. We study the problems of speculative replication and caching in bandwidth- and space-constrained environments for cooperative and stand-alone caches. The methodology relies on simple theoretical models and simulation-based experiments that abstract away system details in return for results with stronger guarantees. The main algorithmic contributions of this dissertation are as follows:

1. We develop *long-term prefetching*, a strategy for determining which objects to speculatively replicate at a large cache constrained by bandwidth or storage space in order to minimize response times. Long-term prefetching and the associated object selection criterion, Goodfetch, take into account both the access rate and the update rate of an object to determine its prefetch-worth. Long-term prefetching is useful for specualatively replicating Web objects at large proxies and content distribution servers, as is demonstrated by our simulation experiments with real proxy traces. The long-term prefetching strategy and associated technical results were first published in "Proceedings of the Sixth International Web Caching and Conten Distribution (WCW '01)" and were presented at Boston, MA, USA in July 2001, and subsequently published in the Computer Communications Journal, Volume 25(4), in 2002.

2. Next, we extend long-term prefetching to a distributed cooperative environment. We consider the problem of how to speculatively place objects in a set of distributed cooperative caches in a hierarchical network where bandwidth to the caches is the constraint and Goodfetch values are known for every ob-

ject at every cache. We develop an object placement algorithm that is shown to be within a constant factor of the optimal. We then extend this algorithm, maintaining asymptotic optimality, to more dynamic scenarios where object access patterns and the universe of objects are not fixed a priori. These results were previously published in "Proceedings of the Twentieth Symposium on Principles of Distributed Computing (PODC '01)" and were presented at Newport, RI in August 2001.

3. Finally, we consider the theoretically intriguing problem of distributed cooperative caching in hierarchical networks in a more traditional space-constrained environment where no a priori information about access patterns is given. We show a non-constant lower bound for the competitive ratio of any online hierarchical cooperative caching algorithm that is given at most a constant factor space advantage. We then present a simple extension of the LRU algorithm to hierarchical networks called HLRU, that when given a blowup of a factor equal to the depth of the hierarchy in the capacity of each cache, lies within a constant factor of an optimal algorithm that has complete knowledge of the request sequence. These results were previously published in "Proceedings of the Sixteenth ACM Symposium on Parallel Algorithms and Architectures (SPAA '04)" and were presented at Barcelona, Spain in July 2004.

The organization of this dissertation is as follows. Chapters 2–4 develop the mechanism contributions of the dissertation. Chapter 2 presents the promise and challenges of ASR. Chapter 3 describes TCP Nice, a mechanism to prevent network interference by providing a background transfer abstraction. Chapter 4 discusses Mars, a self-tuning architecture that enables large-scale replicated systems to incorporate ASR, and presents a case study of an instantiation of Mars, namely, NPS, a non-interfering and easily deployable Web prefetching system. Chapters 5–7 present the algorithmic contributions of the dissertation. Chapter 5 presents

the long-term prefetching strategy and the associated object selection criterion, Goodfetch, that determines what objects to speculatively replicate in a bandwidth-constrained cache. Chapter 6 presents an algorithm for speculative placement of object replicas in hierarchically-organized distributed cooperative caches that are bandwidth-constrained and Goodfetch values of each object at each cache is given. Chapter 7 considers the problem of caching in hierarchically-organized distributed cooperative caches that are constrained by storage space, and proves a lower bound for any online cooperative caching algorithm that is allowed a space advantage at each cache of at most a constant factor over the optimal. We summarize the main contributions and lessons learnt from this dissertation in Chapter 8.

# Chapter 2

# ASR: Promise and Challenges

This section motivates ASR by presenting the vision of an information world where massive-scale replication forms the foundation for pushing the limits of performance and availability and providing richer network services than are offered today. Ideally, ASR can enable near-instantaneous delivery of content on demand, make access to content as available as, say, power supply despite typical wide-area network path connectivity being limited to only 99% of availability (approximately 14 minutes of downtime per day), provide enhanced guarantees on the quality of network services, and provide benfits of multicast by leveraging location-independent addressing of content. All these benefits can be potentially obtained without compromising on the consistency guarantees required for correct operation of an application. As a mechanism, ASR takes us closer to an ideal scenario where performance, availability, and service quality are limited only by available hardware resources and appropriate policy design in the system.

The need for improving performance, availability and service quality is motivated by network applications in use today. The increasing number of devices in use to access data and services, and the increasing importance of the data and services available electronically, both favor "access-anywhere" network-delivered services.

Such services place a high premium on high availability and low response times. Even though end-servers or service-hosting sites advertise an availability of "four nines" (99.99%) or "five nines" (99.999%), the end-to-end service availability (as perceived by clients) is typically limited to two nines because of poor wide-area network availability [38]. Moreover, although network bandwidths are improving quickly, network latencies are much more difficult to improve in wide-area networks, which limits performance for access-anywhere services if those services are delivered from a single location.

## 2.1 Speculative Replication Benefits

Several large-scale distributed systems already use speculative replication to improve performance. A number of previous studies have shown that speculative replication for the Web, or prefetching, can result in significant reductions in end-to-end latency [50, 64, 86, 89, 111, 114, 140, 174]. Padmanabhan and Mogul [140] show that a 30-50% improvement in the access time to fetch a document can be achieved by predictive prefetching, using a Markov model to capture dependencies between accesses. Duchamp obtains a 52% reduction in access time by prefetching hyperlinks in the page being viewed by a user. Our study for a Web proxy shows that prefetching could reduce miss rates by as much as 70% by blowing bandwidth usage by a factor of 2 and disk space by about 25% compared to purely demand-based resource usage. Roussopoulos [156] shows that a controlled update propagation strategy can reduce cache miss costs by a factor of 2 to 200 over just caching with expiration times. Commercial content distribution services like Akamai [5] work by prefetching content at caches situated closer to the client.

Speculative replication has also been shown to improve availability of WAN services. Chandra et al. [38] show an order of magnitude reduction in unavailability for some services by enabling operation in disconnected mode by replicating mo-

bile code and data speculatively on clients. Such benefits in availability are not restricted to web applications with relaxed or no consistency guarantees. Yu [188] shows similar reductions in unavailability by increasing the scale of replication and aggressively propagating updates for a wide range of stronger consistency requirements. Nayate et al. [135] show that for dissemination services, where writes are centralized and reads are distributed across replicas, aggressive update propagation can mask several thousands of seconds of network failures and give a threefold performance improvement at a modest additional bandwidth expenditure of 40% while maintaining strict sequential consistency semantics. In a case study of an e-commerce application, Gao [76] et al. show that a replicated edge-service architecture can mask network failures that render the central database inaccessible 15% of the time and also achieve a five to nine fold improvement in performance over traditional architectures by slightly relaxing consistency within levels acceptable to such an application.

## 2.2 The ASR Vision

Equipping large-scale distributed systems with ASR can benefit access-anywhere services in the following ways:

- *Availability*: ASR can improve availability by speculatively placing a copy of the data or service being accessed within a network accessible by the user. Unlike centrally located services, a service can continue to be available in spite of unavailability of some routes in the network. Studies on overlay networks suggest that application-level routing [9, 160] can significantly improve network availability over the underlying Border Gateway Protocol (BGP) that runs in routers today. The increasing number of devices and networks that users transition across increases the likelihood that only some parts of a wide-area network are accessible while others are not. Such a trend suggests that

massively replicating important services can significantly increase the probability that at least one replica is reachable. Even if the network is entirely unavailable, certain applications may be amenable to operation in a disconnected mode [38] if the entire application and associated data are speculatively replicated locally. Speculative replication is the only way we know to make a service available despite the "primary" provider being unreachable.

- *Response Time*: Applications with real-time guarantees like streaming video, stock quotes, online games etc. require low latencies for an acceptable quality of operation. However, latencies over wide-area networks are fundamentally limited by barriers imposed by the speed of light. The only way we know to reduce such latencies is to speculatively replicate the service to a location near the user before she accesses it. For Web applications, browsers equipped with ASR can keep frequently used bookmarks always refreshed. With shared caches across large communities, and algorithms to predict which hyperlinks will be followed by a user [64], ASR paves a way for finding all content at near-zero response times.

- *Service Quality*: Several applications require quality of service (QoS) guarantees [176] such as lower bounds on bandwidth and upper bounds on loss rate, jitter etc. Proposals to provide QoS guarantees have faced significant deployment barriers as they often require fundamental changes to the architecture of today's networks. ASR can provide enhanced QoS guarantees without changing the already deployed network infrastructure. One could speculatively replicate a service so that it is reachable via a "good" path that meets the required guarantees with high probability. In fact, recent research on probabilistically providing QoS guarantess using overlay networks [166] affirms the claim that such guarantees could also be provided by replicating the service close to the location of the overlay nodes. In fact, replication and overlay-based tech-

niques are complementary and can co-exist to provide QoS guarantees with high probability in existing wide-area networks.

- *Location Independence*: Recent research on using distributed hash tables [165, 150] for content addressibility [44] and several other studies in the last decade point to the benefits of making services location-independent [171]. In today's networks, a service is identified by a URL (like http://www.cnn.com) that rigidly couples the service to its host as well as its network location. The disposition of the scientific community to move to an architecture where services are first-class network citizens rather than being tied to a host and IP address is in keeping with the benefits of moving and replicating services. Replication and location-independence of services thus go hand-in-hand.

- *Multicast*: Multicast is a mechanism for scalable dissemination of data to a group of receivers [61]. Massive replication of services coupled with appropriate service discovery algorithms that locate the "best" replica naturally organize users at the ends of a multicast tree. For example, in a large organization like a university, static content need be retrieved over wide-area links exactly once, for the first user to access it. Subsequent users can retrieve it over a local area network. With falling costs of storage, it is becoming increasingly feasible and economically viable to never discard any piece of information retrieved over a wide-area link [80]. In Bell and Gray's position paper [83] on digital immortality, they point out that rerocrding every conversation a person has ever heard with adequate quality requires less than a terabyte. If one takes into account the amount of content that is shared between users in a community constituting a large organization like a university, a few hundred terabytes of storage, today, is sufficient to cache all digital content accessed, till it expires, at a cost easily affordable by such an organization.

The above claims presenting massive-scale replication as a panacea to today's network ills have several limitations. Massive replication of content in a location-independent manner faces issues related to trust and security. For instance, it may not be feasible or desirable for an enterprise to replicate sensitive content at untrusted locations. Guaranteeing acceptable levels of freshness of all replicas becomes more complex. Websites have an economic incentive to have a client visit them to keep track of advertising revenue or maintain client profiles. In practice, leveraging benefits of multicast, location independence, and smart service discovery algorithms faces significant deployment hurdles in today's Internet. These issues lie beyond the scope of this dissertation and we adopt the stand that enabling mechanisms for ASR to improve response time and availability is the first step towards the broader vision of using massive-scale replication for near-instantaneous content delivery, 100% availability, and enhanced service quality at acceptable hardware costs of replication.

## 2.3 Consistency in ASR Systems

It is easy to discern performance and availability benefits of ASR for wide-area network applications that have weak or no consistency requirements. However, ASR can potentially give significant improvements in response time and availability even for applications that have strict consistency requirements, including applications that require strong consistency (also known as *linearizability*). To understand this claim, consider the metrics in the general replication problem introduced in section 1, namely, response time, availability, consistency, and replication cost. It is a truism that pervades all instances of the replication problem that aggressively replicating content and propagating updates to the fullest extent allowed by bandwidth, computing, storage, and connectivity constraints improves response time and availability. An arbitrary level of strict consistency semantics demanded by

an application can continue to be maintained as consistency information is essentially meta-information whose propagation is orthogonal to propagation of content and updates. *Separation of consistency information from object updates allows us to leverage benefits of ASR without compromising on consistency.* For example, consider a replicated system over a WAN that requires linearizability for correct operation. Whenever there is an update by any node in the system, the replicas can execute a consensus protocol such as Paxos [118, 35] by exchanging only invalidation information of a fixed small size and not the actual update itself. Strong consistency guarantees are maintained by never serving content that has been invalidated at a replica.

It is indeed conceivable that a replica receives invalidation information for some content and, before it can receive the corresponding update, the network gets partitioned in such a manner that no replica possessing the corresponding update is reachable. In such a case, the replica remains unavailable till the corresponding update is received. However, recalling the CAP impossibility argument introduced in section 1, such unavailability is inevitable – a replicated system cannot provide strict consistency guarantees, high availability, and still tolerate arbitrary network partitions. Separating consistency or invalidation information from actual updates combined with ASR mechanisms however allows us to potentially maximize performance and availability of replicated systems for a given set of resource constraints without compromising on consistency. The gap between the level of performance and availability attainable in practice and the theoretical maximum values is strictly a policy issue and essentially a function of the accuracy of the prediction algorithms used.

The arguments presented above are informal. Nayate et al. [135] present a more rigorous argument that shows that sequential consistency can be provided in a single-write replicated system by separating consistency and update information;

coupled with ASR mechanisms, they show significant improvements in performance and availability with modest increases in resource usage. The PRACTI replication work by Dahlin et al., for which the work in this dissertation is one of the geneses, extends the separation of consistency and update information to the more general multi-writer scenario while maintaining causal consistency, and discusses ways of providing stronger levels of consistency guarantees.

## 2.4 Feasibility of ASR

The core argument for enabling system support for ASR mechanisms is that current systems are sacrificing significant potential improvements in performance and availability due to the lack of such support. Tackling the challenges involved in leveraging benefits of ASR at the policy level without appropriate mechanisms is complex (and in some cases infeasible), ineffcient, and risky. However, the benefits of ASR do not come for free; the increased resource usage results in an increased cost of replication. We argue below that such costs represent a worthwhile trade-off and will be further mitigated with current technology trends and workload characteristics.

### Technology Trends

Hardware capacities, i.e. network bandwidth, storage space, and computing power, for a given cost are increasing exponentially with time. Moore's Law for increasing transistor density with time thus extends to bandwidth, storage, and computing power, each with a suitably scaled factor. Figure 2.1 shows the exponential increase of optical channel capacities with time in the last decade. Figure 2.2 shows a similar historical trend for cost per megabyte of hard drives. The data for both of these graphs was collated by the author based on reported optical channel capacities and hard disk costs on the Web [175]. Processor speeds are increasing by around 50% per year [80]. Though the exponential growth in processor speeds is expected to slow

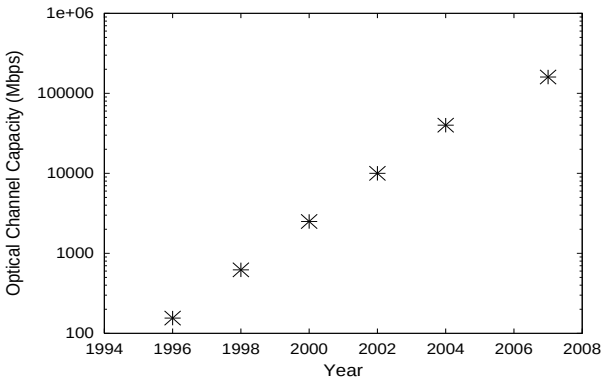



Figure 2.1: Optical channel capacity vs. time

down, Gray [80] suggests that, going by reasonable projections of telecom prices and Moore's Law, cost of computing will continue to be significantly dominated by the cost of network transport. In fact, computing today is essentially free to users as evidenced by several large online services[91, 183] being supported by advertising alone. In addition, emerging Grid [96] technologies will provide mechanisms to further harness massive computing power at low costs.

### Human Time

Human time is valuable and its value is inherently non-decreasing in nature. Increasing capacities and decreasing costs of hardware resources make it more attractive to trade-off such costs for savings in human time. The costs and metrics involved in the general replication problem are not directly comparable. However, Gray and Shenoy [82] suggest a methodology for converting all the units of measurement into their monetary values. They consider the problem of estimating whether caching a data object is economically justified by comparing the cost of storage against the cost of network bandwidth and human waiting time. Chandra et al. [37] extend that methodology to estimate the economic viability of speculatively replicating content

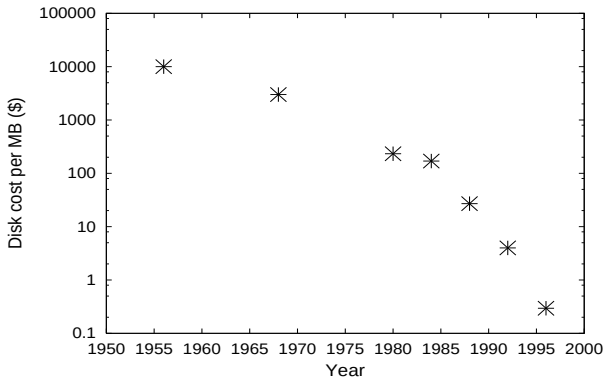



Figure 2.2: Disk cost per MB vs. time

across a wide-area link. Their back-of-the-envelope calculation, reproduced in section 4, suggests that it is economically worthwhile to speculatively replicate content even if there is a one-in-a-hundred chance of it being accessed. In the context of Web prefetching over broadband links, these numbers make a case for expending network and disk resources to prefetch roughly a hundred documents for every document actually accessed by a user. Decreasing costs of hardware resources further strengthen the case for more aggressive prefetching in the future.

### Workload Characteristics

While on one hand, arguments based on economic incentives and technology trends suggest that networks and service providers will be pushed to scale capacities to much higher levels to accomodate the demand for ASR, on the other, systems are already provisioned with considerable extra capacity. Such over-provisioning is a consequence of the burstiness of workloads that can cause spikes in demand much above the average value. Figure 2.3 shows the request load on an IBM server hosting a major sporting event during 1998 averaged over 1-second and 1-minute intervals [37]. Server loads are bursty at many different time-scales with significant differences

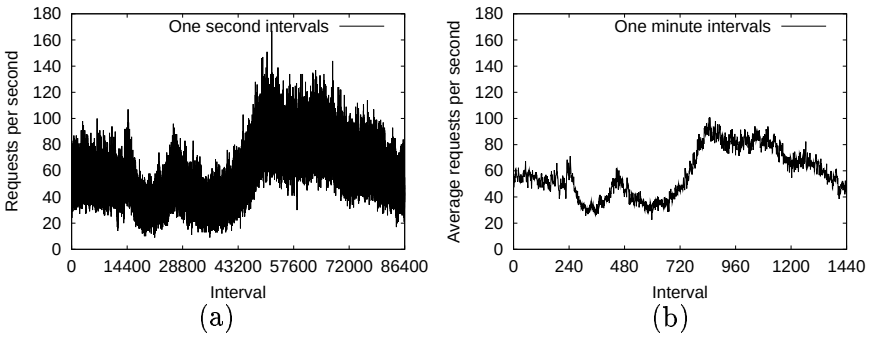
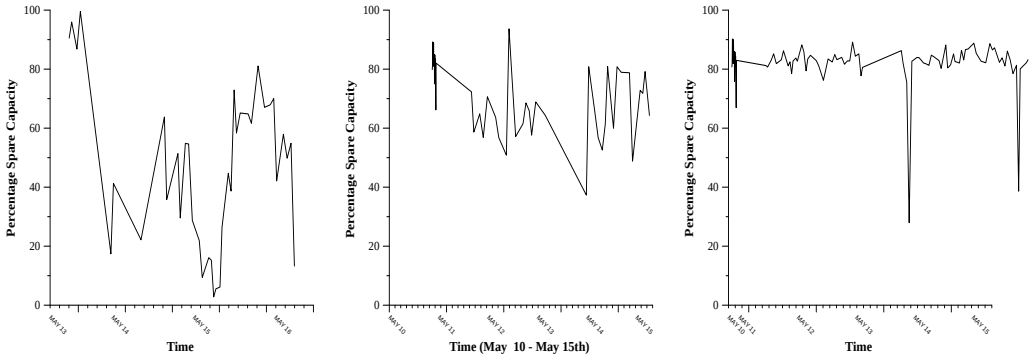



Figure 2.3: Server loads averaged over (a) 1-second and (b) 1-minute interval

between peak and trough loads. Figure 2.4 taken from [110] shows the fraction of unused capacity through a one week period in May 2002 in some links of a commercial cable modem network, the university network Abilene, and a trans-atlantic network respectively, suggesting the presence of significant amounts of spare capacity in these networks. Similar patterns have been noted elsewhere for both servers [42] and networks [1]. The burstiness of workloads and over-provisioning of systems suggest that spare capacity can often be used to support ASR. Conversely, systems built with the capacity to support ASR will also benefit from an increased ability to handle large bursts of load.

## 2.5 ASR Challenges

In spite of the benefits described above, few engineers today incorporate ASR into large-scale replicated systems. Web systems seldom use aggressive prefetching to improve response time. While it is true that network and server capacities are not, at present, sufficient to start prefetching Web objects even if there is a one-in-a-hundred chance of accessing it, such systems do not even utilize the abundant spare capacity already available. The underlying reason is a hidden cost of the complexity




(a) Cable Modem (b) Trans-atlantic Link (c) Abilene

Figure 2.4: Large flow transfer performance over time.

of performing resource management in ASR-enabled systems. The resource management challenges arise from having to balance the following concerns in systems trying to incorporate ASR:

### Interference

Interference refers to competition for resources between speculative load and regular on-demand load. Such competition could arise for any resource such as computing, memory, disk, network bandwidth and could happen at any point in the system such as a server, network, or a client. Interference could result in a net effect of degrading system performance. To understand this claim, consider the effect of speculative replication on average response time. Assume that the system consists of exactly two nodes – a server and a client, where the client requests documents from the server across a wide-area network. Assume that all documents are of equal size, and let $D$ represent the average response time for each document (the time it takes to transfer the entire document) assuming no other traffic in the network and at the server. Now suppose that this client starts prefetching one document for each document she requests on-demand for improving response time. If a document has already been

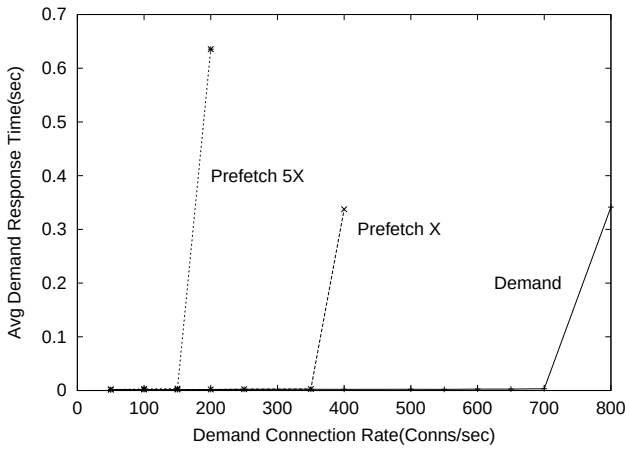



Figure 2.5: Effect of prefetching on demand response times for varying levels of aggressiveness of prefetching

prefetched before she requests it, the corresponding response time to retrieve it from the local cache, or the *hit* latency, is negligible. Assume that the prediction algorithm employed by her browser results in one out of every five prefetched documents to be actually useful, i.e. a request for them appears before they expire in the local cache. Without appropriate mechanisms to prevent interference, prefetch and demand traffic will share the network link equally, resulting in the transfer time of each document to become $2D$; we call this the *miss* latency. We refer to the fraction of hit and miss requests as the *hit rate* and *miss rate* respectively. The average response time with prefetching can now be computed as follows:

$$\begin{aligned} \textit{Average response time} &= \textit{Hit latency} \times \textit{Hit rate} + \textit{Miss latency} \times \textit{Miss rate} \\ &= 0 \times \frac{1}{5} + 2D \times \frac{4}{5} \\ &= 1.8D \end{aligned}$$

Even in this simplistic scenario, we see that interference almost doubles the average response time. In a more realistic setting, consisting of several hundreds of clients across a large-scale network, such interference can cause unpredictable

performance degradation. In a wide-area network, interference can also result in an unfair allocation of resources – speculative load enhancing one user's performance may interfere with and therefore degrade another user's performance. On the other hand, a mechanism for speculative replication in a non-interfering manner, i.e. one that uses network, server, and client resources only when on-demand traffic is not using it, will not cause miss latencies to increase. Thus, average response times decrease when the hit rate is high, i.e. when the prediction algorithm is good and there is sufficient spare capacity, and are protected from increasing otherwise.

### Utilization

A related but conflicting challenge is that of ensuring utilization of available resources for procuring benefits through ASR. The bursty nature of workloads dictates that in order to leverage spare capacity for ASR, the mechanism must be sufficiently responsive at fine time-scales to dynamically increase the aggressiveness of speculative replication when the demand load on the resource is low and vice-versa. An overly conservative ASR mechanism may fail to give benefits that justify the overhead of incorporating ASR mechanisms in the first place, and an overly aggressive mechanism, or one that is not sufficiently responsive, may end up causing interference with demand load.

### Robustness

Networks, servers, and operating systems behave like a complex queueing system where the service rate depends on the arrival rate of requests and the system loses requests beyond a maximum queue size. Initially the service rate increases with the arrival rate till the arrival rate approaches total system capacity after which point the service rate starts to degrade with further increase in the arrival rate. Consequently, in such systems, the response time, as a function of the load, remains low until the

load starts to approach system capacity beyond which point the response times start to rise sharply. It is undesirable for systems to operate in this unstable region of the load-vs-response-time curve. Examples of such scenarios include (i) extreme congestion in the network causing TCP's retransmission timer to keep going off causing exponential backoff, and (ii) extreme load on a web server, resulting in the operating system thrashing.

ASR, if not performed carefully, can drive large-scale systems into unstable regimes. We illustrate this point via two simple experiments. Figure 2.5 shows the response time of requests on a Web server as a function of demand load competing with different levels of prefetching; the number labeling each line approximately represents the number of prefetched requests for each demand request. Increasing the aggressiveness of prefetching gives improvements in response time (not shown in the figure) till the demand load exceeds the knee of the curve corresponding to that level of prefetching, after which point the system becomes unstable. This experiment is forward-referenced here only for the purposes of illustrating the risks associated with ASR. A detailed description of the experimental setup appears in section 4.

In a related experiment, we demonstrate a similar risk-benefit trade-off in prefetching through a scenario where network link capacity is the bottleneck. The experiment consists of sixteen clients accessing Web documents across a broadband network. Figure 2.6 shows three cases where the clients do no prefetching, conservative prefetching and aggressive prefetching respectively. The experiment demonstrates that conservative prefetching can lead to a factor of 3 improvement in response times. However, aggressive prefetching in the same scenario can lead to a sixfold increase in average response time. Careful selection of the level of aggressiveness is thus required to balance the benefits of prefetching with the risk of overload. The details of the setup for this experiment appear in section 4.

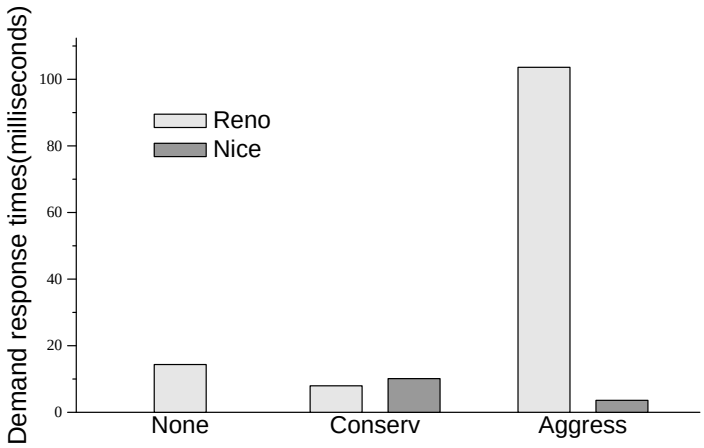



Figure 2.6: Average demand response time vs. aggressiveness of prefetching

### Self-tuning Suport for ASR

Due to inadequate system support for speculative replication, many application designers attempt to balance the above concerns by resorting to *manual-tuning*. In this approach, system parameters are hand-tuned to allocate resources for speculative and regular load in a non-interfering manner. For instance, some Web prefetching systems have been known to use static or workload-specific threshold values to limit the bandwidth consumed by prefetched traffic [64, 140]. A core tenet of this dissertation is that such *threshold-based* approaches to tackle the problem of resource management in large-scale replicated systems are fundamentally flawed. It is the lack of adequate system support for ASR that forces application developers to employ short cuts that are complex in the long run, utilize resources inefficiently, and expose the system to the risk of overload. Hence, a pressing need for large-scale replicated systems today is *self-tuning* system support - mechanisms that render human intervention unnecessary - for ASR. Section 4 elaborates on approaches based on complexity, inefficiency, and risks associated with manual-tuning of system param-

eters, and as an alternative presents Mars, an architecture for self-tuning aggressive speculative replication .

# Chapter 3

# Background Network Transfers

In this chapter, we present a mechanism to prevent *network interference* between speculative and regular traffic. In a large-scale system performing ASR, intereference can occur for any resource at various points in the system such as the server, client or in the network. However, tackling the problem of interference in the network is particularly hard as a WAN is typically a massive, complex system shared by hundreds of thousands of users and controlled by several different administrative domains.

We provide a solution to the problem of network interference by developing a mechanism that provides the abstraction of a *background transfer*. A background transfer refers to a non-interactive transfer that human beings are not waiting for. Such a transfer is typically non-deadline-critical, i.e. there is considerable flexibility in its completion times. An example is the automatic delivery of software updates – a typical user does not actively wait for such transfers to complete and continues to perform other interactive tasks that require greater attention.

Many distributed applications can make use of large background transfers to improve service quality. For example, a broad range of applications and services such as data backup [94], prefetching [174], enterprise data distribution [67], Inter-

net content distribution [4], and peer-to-peer storage [54, 157] can trade increased network bandwidth consumption and possibly disk space for improved service latency [50, 64, 86, 114, 140, 174], improved availability [39, 188], increased scalability [4], stronger consistency [188], or support for mobility [92, 146, 168]. Many of these services have potentially unlimited bandwidth demands where incrementally more bandwidth consumption provides incrementally better service. For example, a Web prefetching system can improve its hit rate by fetching objects from a virtually unlimited collection of objects that have non-zero probability of access [28, 38] or by updating cached copies more frequently as data changes [46, 174, 172].

Current operating systems and networks do not provide good support for aggressive background transfers. In particular, because background transfers compete with foreground requests, they can hurt overall performance and availability by increasing network congestion. Applications must therefore carefully balance the benefits of background transfers against the risk of both *self-interference*, where applications hurt their own performance, and *cross-interference*, where applications hurt the performance of other applications.

Our goal is for the operating system to manage network resources in order to provide a simple abstraction of zero-cost background transfers. A self-tuning background transport layer will enable new classes of applications by (i) simplifying applications, (ii) reducing the risk of being too aggressive, and (iii) making it easier to reap a large fraction of spare bandwidth to gain the advantages of background transfers. Self-tuning resource management seems essential for coping with network conditions that change significantly over periods of seconds (e.g. changing congestion [192]), hours (e.g. diurnal patterns), and months (e.g. technology trends [37, 139]). We focus on managing network resources rather than processors, disks, and memory, both because other work has provided suitable end-station schedulers for these local resources [38, 79, 124, 143, 161], and because networks

are shared across applications, users, and organizations and therefore pose the most critical resource management challenge to aggressive background transfers.

Our system, TCP Nice, dramatically reduces the interference inflicted by background flows on foreground flows. It does so by modifying TCP congestion control to be more sensitive to congestion than traditional protocols such as TCP Reno [103] or TCP Vegas [27] by detecting congestion earlier, reacting to it more aggressively, and allowing much smaller effective minimum congestion windows. Although each of these changes is simple, the combination is carefully constructed to provably bound the interference of background flows on foreground flows while still achieving reasonable throughput in practice. Our Linux implementation of Nice allows senders to select Nice or standard Reno congestion control on a connection-by-connection basis, and it requires no modifications at the receiver .

Our goals are to minimize damage to foreground flows while reaping a significant fraction of available spare network capacity. We evaluate Nice against these goals using theory, microbenchmarks, and application case studies.

Because our first goal is to avoid interference regardless of network conditions or application aggressiveness, our protocol must rest on a sound theoretical basis. In Section 3.2, we argue that our protocol is always less aggressive than Reno, and we prove under a simplified network model that Nice flows interfere with Reno flows' bandwidth by a factor that falls exponentially with the size of the buffer at the bottleneck router independent of the number of Nice flows in the network. Our analysis shows that all three features described above are essential for bounding interference.

Our microbenchmarks comprise both *ns* [138] simulations to stress-test the protocol and Internet measurements to examine the system's behavior under realistic conditions. Our simulation results in Section 3.3 indicate that Nice avoids interference with Reno or Vegas flows across a wide range of background transfer

loads and spare network capacity situations. For example, in one microbenchmark, 16 Nice background flows slow down the average demand document transfer time by less than 10% and reap over 70% of the spare network bandwidth. But in the same situation, 16 backlogged Reno (or Vegas) flows slow demand requests by more than an order of magnitude.

Our Internet microbenchmarks in Section 3.4 measure the performance of simultaneous foreground and background transfers across a variety of Internet links. We find that background flows cause little interference to foreground traffic: the foreground flows' average latency and bandwidth are little changed between when foreground flows compete with background flows and when they do not. Furthermore, we find that there is sufficient spare capacity that background flows reap significant amounts of bandwidth throughout the day. For example, during most hours Nice flows between London, England and Austin, Texas averaged more than 80% of the bandwidth achieved by Reno flows; during the worst hour observed they still saw more than 30% of the Reno flows' bandwidth.

Finally, our case study applications seek to examine the end-to-end effectiveness, the simplicity, and the usefulness of Nice. We examine two services. First, we implement a HTTP prefetching client and server and use Nice to regulate the aggressiveness of prefetching. Second, we study a simplified version of the Tivoli Data Exchange [67] system for replicating data across large numbers of hosts. In both cases, Nice allows us to (i) simplify the application by eliminating magic numbers, (ii) reduce the risk of interfering with demand transfers, and (iii) improve the effectiveness of background transfers by using significant amounts of bandwidth when spare capacity exists. For example, in our prefetching case study, when applications prefetch aggressively, they can improve their performance by a factor of 3 when they use Nice, but if they prefetch using TCP-Reno instead, they overwhelm the network and increase total demand response times by more than a factor of six .

The primary limitation of our analysis is that we evaluate our system when competing against Reno and Vegas TCP flows, but we do not systematically evaluate it against other congestion control protocols such as equation-based [74] or rate-based [151]. Our protocol is strictly less aggressive than Reno, and we expect that it causes little interference with other demand flows, but future work is needed to provide evidence to support this assertion. A second concern is incentive compatibility: will users use low priority flows for background traffic when they could use high priority flows instead? We observe that most of the "aggressive replication" applications cited above do, in fact, voluntarily limit their aggressiveness by, for instance, prefetching only those objects whose priority of use exceeds a threshold [64, 174]. Two factors may account for this phenomenon. First, good engineers may consider the social costs of background transfers and therefore be conservative in their demands. Second, most users have an incentive to at least avoid self-interference where a user's background traffic interferes with that user's foreground traffic from the same or different application. We thus believe that Nice is a useful tool for both responsible and selfish engineers and users.

The rest of this chapter proceeds as follows. Section 3.1 describes the Nice congestion control algorithm. Sections 3.2, 3.3, and 3.4 present our analytic results, NS microbenchmark results, and Internet measurement results respectively. Our experience with case study applications follows in Section 3.5. Finally, Section 3.6 puts this work in context with related work, followed by conclusions in Section 3.8.

## 3.1 Design and Implementation

In designing our system, we seek to balance two conflicting goals. An ideal system would (i) cause no interference to demand transfers and (ii) consume 100% of available spare bandwidth. In order to provide a simple and safe abstraction to applications, we emphasize the former goal and will be satisfied if our protocol

makes use of a significant fraction of spare bandwidth. Although it is easy for an adversary to construct scenarios where Nice does not get any throughput in spite of there being sufficient spare capacity in the network, our experiments confirm that in practice, Nice obtains a significant fraction of the throughput of Reno or Vegas when there is spare capacity in the network.

### 3.1.1 Background: Existing Algorithms

Congestion control mechanisms in existing transmission protocols are composed of a *congestion signal* and a *reaction policy*. The congestion control algorithms in popular variants of TCP (Reno, NewReno, Tahoe, SACK) use packet loss as a congestion signal. In steady state, the reaction policy uses additive increase and multiplicative decrease (AIMD) in which the sending rate is controlled by a congestion window that is multiplicatively decreased by a factor of two upon a packet drop and is increased by 1 per window of data acknowledged. The AIMD framework is believed to be fundamental to the robustness of the Internet [45, 103].

However, with respect to our goal of minimizing interference, this congestion signal – a packet loss – arrives too late to avoid damaging other flows. In particular, overflowing a buffer (or filling a RED router enough to cause it to start dropping packets) may trigger losses in other flows, forcing them to back off multiplicatively and lose throughput.

In order to detect incipient congestion due to interference, we monitor round-trip delays of packets and use increasing round-trip delays as a signal of congestion. In this respect, we draw inspiration from TCP Vegas [27], a protocol that differs from TCP-Reno in its congestion avoidance phase. By monitoring round-trip delays, each Vegas flow tries to keep between $\alpha$ (typically 1) and $\beta$ (typically 3) packets buffered at the bottleneck router. If fewer than $\alpha$ packets are queued, Vegas increases the window by 1 per window of data acknowledged. If more than $\beta$ packets are queued,

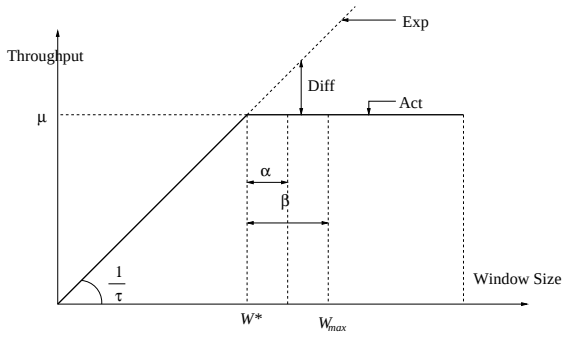



Figure 3.1: TCP Vegas congestion avoidance

the algorithm decreases the window by 1 per window of data acknowledged. Vegas adjusts the window $W$ once every round as follows ($minRTT$ is the minimum value of all measured round-trip delays and $observedRTT$ is the round-trip delay experienced by a distinguished packet in the previous round):

$E \leftarrow \frac{W}{minRTT}$ // Expected throughput

$A \leftarrow \frac{W}{observedRTT}$ // Actual throughput

$Diff \leftarrow (E - A) \cdot minRTT$

**if** $(Diff < \alpha)$
    $W \leftarrow W + 1$
**else if** $(Diff > \beta)$
    $W \leftarrow W - 1$

As figure 3.1 shows, bounding the difference between the actual and expected throughput translates to maintaining between $\alpha$ and $\beta$ packets in the bottleneck

router. Although Vegas seems a promising candidate protocol for background flows, it has some drawbacks: (i) Vegas has been designed to compete for throughput approximately fairly with Reno, (ii) Vegas backs off when the number of queued packets from its flow increases, but it does not necessarily back off when the number of packets enqueued by other flows increase, and (iii) each Vegas flow tries to keep 1 to 3 packets in the bottleneck queue, hence a collection of background flows could cause significant interference.

Note that even setting $\alpha$ and $\beta$ to very small values does not prevent Vegas from interfering with cross traffic. The linear decrease on the "$Diff > \beta$" trigger is not responsive enough to keep from interfering with other flows. We confirm this intuition using simulations and Internet experiments, and it also follows as a conclusion from the theoretical analysis.

### 3.1.2 TCP Nice

The Nice extension adds three components to Vegas: first, a more sensitive congestion detector; second, multiplicative reduction in response to increasing round-trip times; and third, the ability to reduce the congestion window below 1. These additions are simple, but our analysis and experiments demonstrate that the omission of any of them would fundamentally increase the interference caused by background flows.

A Nice flow monitors round-trip delays, estimates the total queue size at the bottleneck router, and signals congestion when this total queue size exceeds a fraction of the estimated maximum queue capacity. Nice uses $minRTT$, the minimum observed round-trip time, as the estimate of the round-trip delay when queues are empty, and it uses $maxRTT$ as an estimate of the round-trip time when the bottleneck queue is full. If more than *fraction* of the packets Nice sends during a RTT

window encounter delays exceeding $minRTT + (maxRTT - minRTT) \cdot threshold$, our detector signals congestion. Round-trip delays of packets are indicative of the current bottleneck queue size and the threshold represents the fraction of the total queue capacity that starts to trigger congestion. The Nice congestion avoidance mechanism incorporating the *interference trigger* with threshold $t$ and fraction $f$ can be written as follows ($curRTT$ is the round-trip delay experienced by each packet):

per ack operation:

**if** $(curRTT > (1 - t) \cdot minRTT + t \cdot maxRTT)$

$numCong{+}{+}$;

per round operation:

**if** $(numCong > f \cdot W)$

$W \leftarrow W/2$

**else** {

... // Vegas congestion avoidance follows

}

If the congestion condition does not trigger, Nice falls back on Vegas' congestion avoidance rules. If a packet is lost, Nice falls back on Reno's rules. The final change to congestion control is to allow the window sizes to multiplicatively decrease below 1, if so dictated by the congestion trigger and response. In order to affect window sizes less than 1, we send a packet out after waiting for the appropriate number of smoothed round-trip delays.

Maintaining a window of less than 1 causes us to lose *ack-clocking*, but the flow continues to send at most as many packets into the network as it gets out. In this phase the packets act as network probes waiting for congestion to dissipate. By allowing the window to go below 1, Nice retains the non-interference property even for a large number of flows. Both our analysis and our experiments confirm

the importance of this feature: this optimization significantly reduces interference, particularly when testing against several background flows. A similar optimization has been suggested even for regular flows to handle cases when the number of flows starts to approach the bottleneck router buffer size [131].

When a Nice flow signals congestion, it halves its current congestion window. In contrast Vegas reduces its window by one packet each round that encounters long round-trip times and only halves its window if packets are lost (falling back on Reno-like behavior.) The combination of more aggressive detection and more aggressive reaction may make it more difficult for Nice to maximize utilization of spare capacity, but our design goals lead us to minimize interference even at the potential cost of utilization. Our experimental results show that even with these aggressively timid policies, we achieve reasonable levels of utilization in practice.

As in TCP Vegas, maintaining running measures of $minRTT$ and $maxRTT$ have their limitations - for example, if the network is in a state of persistent congestion, a bad estimate of $minRTT$ is likely to be obtained. However, past studies [2, 159] have indicated that a good estimate of the minimum round-trip delay can typically be obtained in a short time; our experience supports this claim. The use of minimum and maximum values makes the prototype sensitive to outliers. Using the first and ninety-ninth percentile values could improve the robustness of this algorithm, but we have not tested this optimization. Route changes during a transfer can also contribute to inaccuracies in RTT estimates. However such changes are uncommon [144] and we speculate that they can be handled by maintaining exponentially decaying averages for $minRTT$ and $maxRTT$ estimates.

### 3.1.3 Prototype Implementation

We implement a prototype Nice system by extending an existing version of the Linux kernel that supports Vegas congestion avoidance. Like Vegas, we use microsecond

resolution timers to monitor round-trip delays of packets to implement a congestion detector. In our implementation of Nice, we set the corresponding Vegas parameters $\alpha$ and $\beta$ to 1 and 3 respectively. After the first round-trip delay estimate, maxRTT is initialized to $2 \cdot minRTT$.

The Linux TCP implementation maintains a minimum window size of two in order to avoid delayed acknowledgements by receivers that attempt to send one acknowledgment every two packets. In order to allow the congestion window to go to 1 or below 1, we add a new timer that runs on a per-socket basis when the congestion window for the particular socket is below two. When in this phase, the flow waits for the appropriate number of RTTs before sending two packets into the network. Thus, a window of 1/16 means that the flow sends out two packets after waiting for 32 smoothed round-trip times. We limit the minimum window size to 1/48 in our prototype.

Our congestion detector signals congestion when more than $fraction = 0.5$ packets during an RTT encounter delays exceeding $threshold = 0.2$. We discuss the sensitivity to *threshold* in more detail in Section 3.2. The *fraction* does not enter directly into our analysis; our experimental studies in Section 3.3 indicate that the interference is relatively insensitive to the *fraction* parameter chosen. Since packets are sent in bursts, most packets in a round observe similar round-trip times. In the future we plan to study pacing packets across a round in order to obtain better samples of prevailing round-trip delays.

Our prototype provides a simple API to designate a flow as a background flow through an option in the *setsockopt* system call. By default, flows are foreground flows.

## 3.2 Analysis

Experimental evidence alone is insufficient to allow us to make strong statements about Nice's non-interference properties for general network topologies, background flow workloads, and foreground flow workloads. We therefore analyze it formally to bound the reduction in throughput that Nice imposes on foreground flows. Our primary result is that under a simplified network model, for long transfers, the reduction in the throughput of Reno flows is asymptotically bounded by a factor that falls exponentially with the maximum queue length of the bottleneck router irrespective of the number of Nice flows present.

Theoretical analysis of network protocols, of course, has limits. In general, as one abstracts away details to gain tractability or generality, one risks omitting important behaviors. Most significantly, our formal analysis assumes a simplified fluid approximation and synchronous network model, as described below. Also, our formal analysis holds for long background flows, which are the target workload of our abstraction. But it also assumes long foreground Reno flows, which are clearly not the only cross-traffic of interest. Finally, in our analysis, we abstract detection by assuming that at the end of each RTT epoch, a Nice sender accurately estimates the queue length during the previous epoch. Although these assumptions are restrictive, the insights gained in the analysis lead us to expect the protocol to work well under more general circumstances. The analysis has also guided our design, allowing us to include features that are necessary for noninterference while excluding those that are not. Our experience with the prototype has supported the benefit of using theoretical analysis to guide our design: we encountered few surprises and required no topology or workload-dependent tuning during our experimental effort.

We use a simplified fluid approximation model of the network to help us model the interaction of multiple flows using separate congestion control algorithms. This model assumes infinitely small packets. We simplify the network itself to a

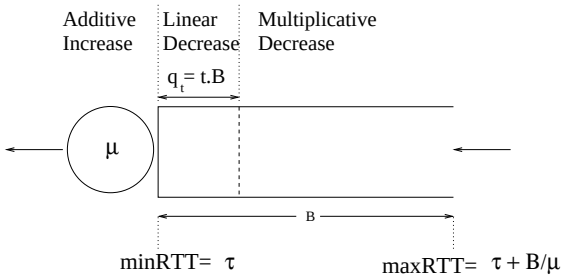



Figure 3.2: Nice Queue Dynamics

source, destination, and a *single bottleneck*, namely a router that performs drop-tail queuing as shown in Figure 3.2. Let $\mu$ denote the service rate of the queue and $B$ the buffer capacity at the queue. Let $\tau$ be the round-trip delay of packets between the source and destination excluding all queuing delays. We consider a fixed number of connections, $m$ following Reno and $l$ following Nice, each of which has one continuously backlogged flow between a source and a destination. Let $t$ be the Nice threshold and $q_t = t \cdot B$ be the corresponding queue size that triggers multiplicative backoff for Nice flows. The connections are homogeneous, *i.e.* they experience the same propagation delay $\tau$. Moreover, the connections are synchronized so that in the case of buffer overflow, all connections simultaneously detect a loss and multiply their window sizes by $\gamma$. Models assuming flow synchronization have been used in previous analyses [23]. We model only the congestion avoidance phase to analyze the steady-state behavior.

We obtain a bound on the reduction in the throughput of Reno flows due to the presence of Nice flows by analyzing the dynamics of the bottleneck queue. We achieve this goal by dividing the duration of the flows into *periods*. In each period we bound the decrease in the number of Reno packets processed by the router due to interfering Nice packets. In the following we give an outline of this analysis.

Let $W_r(t)$ and $W_n(t)$ denote respectively the total number of outstanding

Reno and Nice packets in the network at time $t$. $W(t)$, the total window size, is $W_r(t) + W_n(t)$. We trace these window sizes across periods. *The end of a period and the beginning of the next is marked by a packet loss*, at which time each flow reduces its window size by a factor of $\gamma$. $W(t) = \mu\tau + B$ just before a loss and $W(t) = (\mu\tau + B) \cdot \gamma$ just after. Let $t_0$ be the beginning of one such period after a loss. Consider the case when $W(t_0) = (\mu\tau + B)\gamma < \mu\tau$ and $m > l$. For ease of analysis we assume that the "Vegas $\beta$" parameter for the Nice flows is 0, *i.e.* the Nice flows additively decrease upon observing round-trip times greater than $\tau$. The window dynamics in any period can be split into three intervals as described below.

**I1** - *Additive Increase, Additive Increase*: In this interval $[t_0, t_1]$ both Reno and Nice flows increase linearly. $W(t)$ increases from $W(t_0)$ to $W(t_1) = \mu\tau$, at which point the queue starts building.

**I2** - *Additive Increase, Additive Decrease*: This interval $[t_1, t_2]$ is marked by additive increase of $W_r$, but additive decrease of $W_n$ as the "*Diff* $> \beta$" rule triggers the underlying Vegas controls for the Nice flows. The end of this interval is marked by $W(t_2) = \mu\tau + q_t$.

**I3** - *Additive Increase, Multiplicative Decrease*: In this interval $[t_2, t_3]$, $W_n(t)$ multiplicatively decreases, by a factor $\gamma'$ say, in response to observing queue lengths above $q_t$. However, the rate of decrease of $W_n(t)$ is bounded by the rate of increase of $W_r(t)$, as any faster a decrease will cause the queue size to drop below $q_t$. At the end of this interval $W(t_3) = \mu\tau + B$. At this point, each flow decreases its window size by a factor of $\gamma$, thereby entering into the next period.

In order to quantify the interference experienced by Reno flows because of the presence of Nice flows, we formulate differential equations to represent the vari-

ation of the queue size in a period. We then show that the values of $W_r$ and $W_n$ at the beginning of periods stabilize after several losses, so that the length of a period converges to a fixed value. It is then straightforward to compute the total amount of Reno flow sent out in a period. We then show that the interference $I$, defined as the fractional loss in throughput experienced by Reno flows because of the presence of Nice flows, can be bounded as follows.

**Theorem 1**: The interference $I$ is bounded as

$$I \leq \frac{4m \cdot e^{(-\frac{B(1-t)\gamma}{m})}}{(\mu\tau + B)\gamma} \tag{3.1}$$

The expression for $I$ indicates that all three design features of Nice are fundamentally important for reducing interference. The interference falls exponentially with $B(1-t)$ or $B - q_t$, which reflects the time that Nice has to multiplicatively back off before packet losses occur. Intuitively, multiplicative decrease allows any number of Nice flows to get out of the way of additively increasing demand flows. The dependence on the ratio $\frac{B}{m}$ suggests that as the number of demand flows approaches the maximum queue size the non-interference property starts to break down. This breakdown is not surprising as each flow barely gets to maintain one packet in the queue and TCP Reno is known to behave anomalously under such circumstances [131]. In a well designed network, when $B \gg m$, it can be seen that the dependence on the threshold $t$ is weak, *i.e.* interference is small when $t$ is, and careful tuning of the exact value of $t$ in this region is unnecessary. The full analysis below shows that the above bound on $I$ holds even for the case when $m \ll l$. Allowing window sizes to multiplicatively decrease below one is crucial in this proof. Observe that when $(\mu\tau + B)\gamma > \mu\tau$, Nice flows do not get any throughput. However, this condition implies that the network is already saturated and does not have any spare capacity.

### 3.2.1 Proof of Theorem 1

In this section we derive the upper bound on interference stated in Theorem 3.1. We first consider the case when the number of Reno flows $m > l$ and then show that the derivation also holds true for the case when the $m < l$. We formulate differential equations to describe the dynamics of the queue size as explained in the previous section.

**I1** - *Additive Increase, Additive Increase* $[t_0, t_1]$ :

$$\begin{cases} \frac{dW_r(t)}{dt} &= \frac{m}{\tau} \\ \\ \frac{dW_n(t)}{dt} &= \frac{l}{\tau} \end{cases} \tag{3.2}$$

**I2** - *Additive Increase, Additive Decrease* $[t_0, t_1]$ :

$$\begin{cases} \frac{dW_r(t)}{dt} &= \frac{m\mu}{W(t)} \\ \\ \frac{dW_n(t)}{dt} &= -\frac{l\mu}{W(t)} \quad \text{if } W_n(t) > l, \quad \text{else } 0 \end{cases} \tag{3.3}$$

**I3** - *Additive Increase, Multiplicative Decrease* $[t_0, t_1]$ :

$$\begin{cases} \frac{dW_r(t)}{dt} &= \frac{m\mu}{W(t)} \\ \\ \frac{dW_n(t)}{dt} &= -min(\frac{\gamma^{'} \cdot W_n(t)\mu}{W(t)}, \frac{m\mu}{W(t)}) \end{cases} \tag{3.4}$$

Next we introduce three lemmas that make it easier to calculate the reduction in the total amount of Reno flow that gets sent out in a period due to interference from Nice flows, without actually having to solve equations (3.2), (3.3) and (3.4) completely.

**Lemma 1**: The total amount of flow sent by the Reno flows in a period depends only on the initial and final values of $W_r$ in the period.

*Proof*: The total amount of flow sent out by the Reno flows is given by integrating the instantaneous sending rate over the duration of the period. The instantaneous sending rate at time $t$ is obtained by dividing the current window size $W_r(t)$ by the current value of the round-trip delay $W(t)/\mu$. Thus, the total amount of Reno flow $R$ sent out in a period beginning at $t_0$ is given by:

$$R = \frac{1}{m}\int_{t_0}^{t_0+T} \frac{W_r(t)\mu}{W(t)} dt \tag{3.5}$$

From ( 3.12),( 3.13) and ( 3.14) we observe that the rate of increase of $W_r$ throughout a period is given by:

$$\frac{dW_r(t)}{dt} = \frac{m\mu}{W(t)} \tag{3.6}$$

From ( 3.5) and ( 3.6) we get:

$$\begin{aligned} R &= \frac{1}{m}\int_{W_r(t_0)}^{W_r(t_0+T)} W_r(t) dW_r(t) \\ &= \frac{W_r^2(t)}{2m} \Big|_{t_0}^{t_0+T} \end{aligned} \tag{3.7}$$

Hence proved.

**Lemma 2**: The length of a period $T$ in a system with $m$ Reno flows and a non-zero number of Nice flows is shorter than that of a system consisting of only the Reno flows.

*Proof*: The proof of this lemma follows straighforwardly from the dynamics of $W_r(t)$ alone. Let $T'$ denote the length of a period when only $m$ Reno flows (and no Nice

flows) are present, and $W_r^{'}(t)$ denote the total number of outstanding (Reno) packets at time $t$. Rewriting ( 3.6) for $W_r^{'}(t)$:

$$W_r^{'}(t)dW_r^{'}(t) = m\mu dt \tag{3.8}$$

Assume that a period begins at time $t_0$. Integrating both sides of ( 3.8) and swapping sides we get:

$$\begin{aligned} T^{'} &= \frac{1}{m\mu}\int_{t_0}^{t_0+T^{'}} W_r^{'}(t)dW_r^{'}(t) \\ &= \int_{(\mu\tau+B)\gamma}^{(\mu\tau+B)} W_r^{'}dW_r^{'} \end{aligned} \tag{3.9}$$

Similarly, using ( 3.6) we obtain for $T$:

$$\begin{aligned} T &= \int_{t_0}^{t_0+T} W(t)dW_r(t) \\ &= \int_{t_0}^{t_0+T} W(t)dW(t).\frac{dW_r(t)}{dW(t)} \\ &= \int_{(\mu\tau+B)\gamma}^{(\mu\tau+B)} WdW.\frac{dW_r}{dW} \end{aligned} \tag{3.10}$$

Notice that the multiplicative term $\frac{dW_r}{dW} < 1$ throughout the period, as $W = W_r + W_n$. Therefore $T < T^{'}$.

**Lemma 3**: The residual number of total outstanding Nice packets just before a packet loss in any period is at most $\frac{m}{\gamma}^{'}.\ e^{(-(\frac{\gamma(B-q_t)}{m}))}$

*Proof*: Let $t_0$ mark the beginning of a period just after a packet loss so that $W(t_0) = (\mu\tau + B)\gamma < \mu\tau$. Let $t_1, t_2, t_3$ mark respectively the end of the three intervals constituing the current period. The beginning of the interval $[t_2, t_3]$ initiates multiplicative decrease on part of the Nice flows. The dynamics are given by equation ( 3.4).

If $W_n(t_2) > \frac{m}{\gamma^{'}}$, $W_n$ multiplicatively decreases at a rate just enough to counter the

rate of increase of $W_r$, thereby keeping $W$ fixed at $\mu\tau + q_t$ till time $t_2^{'}$ such that $W_n(t_2^{'}) = 2m$. For the rest of the period $[t_2^{'}, t_3]$ the system dynamics are given by:

$$\begin{cases} \frac{dW_r(t)}{dt} &= \frac{m\mu}{W(t)} \\ \\ \frac{dW_n(t)}{dt} &= -\frac{\gamma^{'} \cdot W_n(t)\mu}{W(t)} \end{cases} \tag{3.11}$$

If $W_n(t_2) < \frac{m}{\gamma^{'}}$, (3.11) completely describes the dynamics throughout the interval $[t_2, t_3]$. With the initial boundary conditions $W_n(t_2^{'}) = \frac{m}{\gamma^{'}}$, $W(t_2^{'}) = \mu\tau + q_t$, and the final boundary condition $W(t_3) = \mu\tau + B$, equation (3.11) yields a unique solution to $W_n(t_3)$ – the number of residual Nice packets in the queue just before a loss – which we also denote by $\delta$.. To compute this value, divide the second differential equation by the first to obtain □

$$\frac{dW_n(t)}{dW_r(t)} = -\frac{\gamma^{'} W_n(t)}{m}$$

$$\Rightarrow \frac{dW_n(t)}{W_n(t)} = -\frac{\gamma^{'} dW_r(t)}{m}$$

Integrating both sides we obtain:

$$\int_{\frac{m}{\gamma^{'}}}^{\delta} \frac{dW_n}{W_n} = -\frac{\gamma^{'}}{m} \int_{\mu\tau + q_t - 2m}^{\mu\tau + B - \delta} dW_r$$

$$\Rightarrow \log(\frac{\gamma^{'} . \delta}{m}) \leq -\frac{\gamma^{'}}{m} \cdot [B - q_t]$$

$$\Rightarrow \delta \leq \frac{m}{\gamma^{'}} \cdot e^{(-(\frac{\gamma^{'} . (B - q_t)}{m}))}$$

Hence lemma 3. □

**Proof of Theorem 1**

Interference is calculated as the fractional loss in throughput obtained by Reno flows due to the presence of Nice flows. The throughput obtained by Reno flows in the

presence of Nice flows is the total Reno flow sent out in a period divided by the length of a period $T$. We first consider the case $m > l$.

**Case 1:** $m > l$

By Lemma 2, the total flow sent out by the Reno flows is a period depends only on the initial and final values of $W_r(t)$ in a period. Thus, the throughput $P$ obtained by Reno flows is computed using ( 3.7) as:

$$P = \frac{1}{T} \cdot \frac{[(\mu\tau+B-\delta)^2-((\mu\tau+B-\delta)\gamma)^2]}{2m}$$

The throughput obtained by the Reno flows in the absence of any Nice flows is given by:

$$Q = \frac{1}{T'} \cdot \frac{(\mu\tau+B)^2(1-\gamma^2)}{2m}$$

The interference $I$ defined as the fractional loss in throughput is given by $\frac{Q-P}{Q}$. By Lemma 3, $T' > T$, which yields:

$$I \leq \frac{[(\mu\tau+B-\delta)^2-(\mu\tau+B)^2]}{(\mu\tau+B)^2}$$

$$\leq \frac{2\delta}{(\mu\tau+B)}$$

$$\leq \frac{2m.e^{(-\frac{B(1-t)\gamma'}{m})}}{(\mu\tau+B)\gamma\gamma'}$$

Substituting $\gamma = 2$ to represent multiplicative *halving* of window, we obtain

$$I \leq \frac{4m.e^{(-\frac{B(1-t)}{2m})}}{(\mu\tau+B)\gamma}$$

**Case 2:** $m < l$

In this case we simply give the differential equations governing the dynamics of

the window sizes claim that Lemmas 1 to 3 hold in this case as well. The verification of the same is left as an exercise to the reader.

In interval $[t_0, t_1]$ $W(t)$ increases from $W(t_0)$ to $\mu\tau$, at which point the queue starts building up. Both Reno and Nice flows increase linearly and their dynamics can be represented as:

$$\begin{cases} \frac{dW_r(t)}{dt} &= \frac{m}{\tau} \\ \\ \frac{dW_n(t)}{dt} &= \frac{l}{\tau} \end{cases} \tag{3.12}$$

The next interval $[t_1, t_2]$ is marked by additive increase of $W_r$, but additive decrease of $W_n$ as the "*Diff* $> \beta$" rule triggers the underlying Vegas controls for the Nice flows. However, the rate of decrease of $W_n(t)$ is bounded by the rate of increase of $W_r(t)$. The two therefore exactly balance each other and the total window size $W(t)$ remains constant at $\mu\tau$. Moreover, in the additive decrease phase, each Nice flow maintains a minimum window of 1, which implies that $W_n(t) \geq l$ in this phase. The round-trip time experienced by each packet when the queue is non-empty is given by $W(t)/\mu$. Thus, the window dynamics during interval $[t_1, t_2]$ are as follows:

$$\begin{cases} \frac{dW_r(t)}{dt} &= \frac{m\mu}{W(t)} \\ \\ \frac{dW_n(t)}{dt} &= -\frac{m\mu}{W(t)}, \quad \text{if } W_n(t) > l \\ &= 0 \quad \text{otherwise} \end{cases} \tag{3.13}$$

The end of this interval is the time $t_2$ when $W(t_2) = \mu\tau + q_t$, where $q_t$ is the threshold queue size that begins multiplicative backoff for Nice flows. However, again the rate of decrease of $W_n(t)$ is bounded by the rate of increase of increase of $W_r(t)$. Thus, the dynamics of interval $[t_2, t_3]$ are governed by:

$$\begin{cases} \frac{dW_r(t)}{dt} &= \frac{m\mu}{W(t)} \\ \\ \frac{dW_n(t)}{dt} &= -min(\frac{W_n(t)\mu}{\gamma'.W(t)}, \frac{m\mu}{W(t)}) \end{cases} \tag{3.14}$$

The end of the above interval marks the completion of the period. At this point $W(t_3) = \mu\tau + B$, and right after, each flow decreases its window size by a factor of $\gamma$, thereby entering into the next period.

Using Lemmas 1 to 3, it can be shown, exactly as in case 1, that the interference bound given in ( 3.1) holds for this case as well.

Hence Theorem 1 follows. □

## 3.3 ns Controlled Tests

The goal of our simulations is to validate our hypotheses in a controlled environment. In particular, we wish to i) test the non-interference property of Nice and ii) determine if Nice gets any useful bandwidth for the workloads considered. By using controlled *ns* [138] simulations in this phase of the study we can stress test the system by varying network configurations and load to extreme values. We can also systematically compare the Nice algorithm against others. Overall, the experiments support our theses:

- Nice flows cause almost no interference irrespective of the number of flows.
- Nice gets a significant fraction of the available spare bandwidth.
- Nice performs better than other existing protocols, including Reno, Vegas, and Vegas with reduced $\alpha$ and $\beta$ parameters.

### 3.3.1 Methodology

We use *ns* 2.1b8a for our simulation experiments. The topology used is a bar-bell in which $N$ TCP senders transmit through a shared bottleneck link $L$ to an equal number of receivers. The router connecting the senders to $L$ becomes the bottleneck queue. Routers perform drop-tail FIFO queueing except in experiments with RED turned on. The buffer size is set to the bandwidth delay product. Packets are 512 bytes in size and the propagation delay is set to 50ms. We vary the capacity of the link in order to simulate different amounts of spare capacity.

We use a 15 minute section of a Squid proxy trace logged at UC Berkeley as the foreground traffic over L. The number of flows fluctuates as clients enter and leave the system as specified by the trace. On average there are about 12 active clients. In addition to this foreground load, we introduce permanently backlogged background flows. For the initial set of experiments we fix the bandwidth of the link to twice the average demand bandwidth of the trace. The primary metric we use to measure interference is the average transfer latency of a document *i.e.*, the time between its first packet being sent and the receipt of the ack corresponding to the last packet. We use the total number of bytes transferred by the background flows as the measure of its utilization of spare capacity.

Unless otherwise specified, the values of the $threshold$ and $fraction$ for Nice are set to 0.1 and 0.5 respectively. We compare the performance of Nice to several other strategies for sending background flows. First, we compare with router prioritization that services a background packet only if there are no queued foreground packets. Router prioritization is the ideal strategy for background flow transmission, as background flows never interfere with foreground flows. In addition, we compare to Reno, Vegas($\alpha = 1, \beta = 3$), Vegas($\alpha = 0, \beta = 0$).

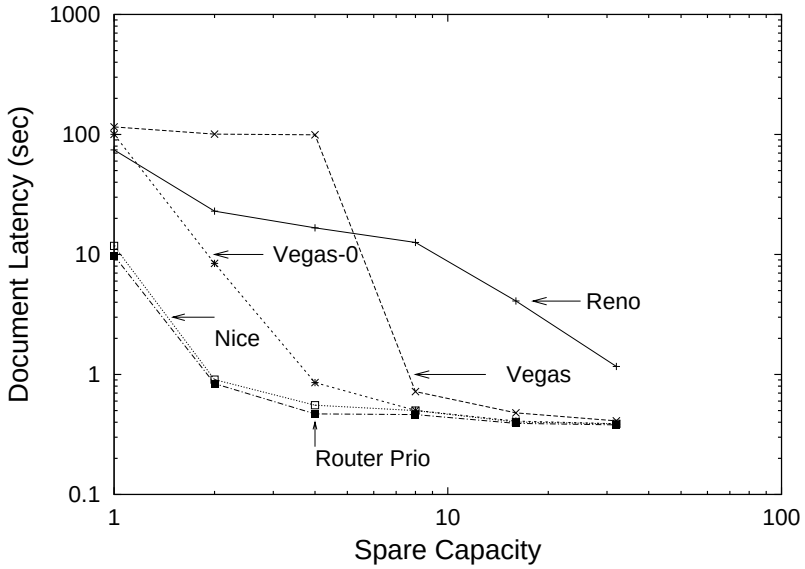



Figure 3.3: Spare capacity vs Latency

### 3.3.2 Results

**Experiment 1:** In this experiment we fix the number of background flows to 16 and vary the spare capacity, $S$. To achieve a spare capacity $S$, we set the bottleneck link bandwidth $L = (1 + S) \cdot averageDemandBW$, where $averageDemandBW$ is the total number of bytes transferred in the trace divided by the duration of the trace. Figure 3.3 plots the average document transfer latency for foreground traffic as a function of the spare capacity in the network. Different lines represent different runs of the experiments using different protocols for background flows. It can be seen that Nice is hardly distinguishable from router prioritization whereas, the other protocols cause a significant increase in foreground latency. Note that the Y-axis is on a log scale, which means that in some cases Reno and Vegas increase foreground document transfer latencies by over an order of magnitude.

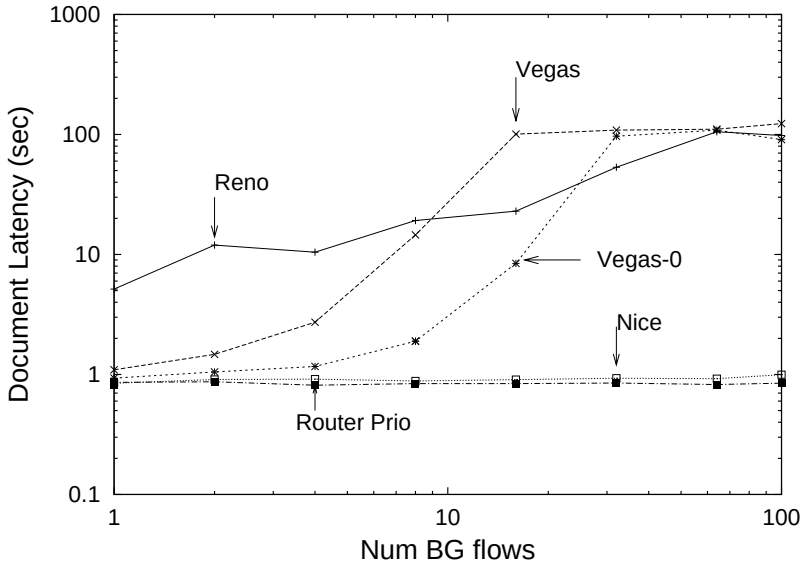



Figure 3.4: Number of BG flows vs Latency

**Experiment 2: Sensitivity to number of BG flows** In this experiment we fix the spare capacity $S$ of the network to 1 and vary the number of background flows. Figure 3.4 plots the latency of foreground document transfers against the number of background flows. Even with 100 background Nice flows, the latency of foreground documents is hardly distinguishable from the ideal case when routers provide strict prioritization. On the other hand, Reno and Vegas background flows can cause foreground latencies to increase by orders of magnitude. Figure 3.5 plots the number of bytes the background flows manage to transfer. A single background flow reaps about half the spare bandwidth available under router prioritization; this background throughput improves with increasing number of background flows but remains below router prioritization. The difference is the price we pay for ensuring non-interference with an end-to-end algorithm. Note that although Reno and Vegas obtain better throughput, even for a small number of flows they go beyond the

router prioritization line, which means they steal bandwidth from foreground traffic. Figure 3.6 shows the number of bytes transferred by the foreground traffic. While Nice hardly hurts the foreground throughput, Reno and Vegas end up decreasing it by upto 60%.

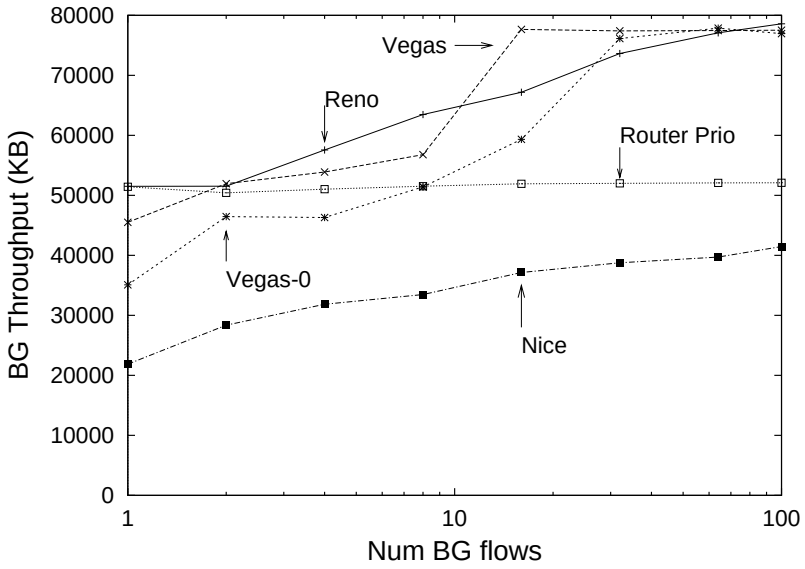



Figure 3.5: Number of BG flows vs BG throughput

The three components of Nice that make it non-interfering are - i) early congestion detection, ii) aggressive multiplicative backoff, and iii) allowing window sizes below 1 relative to regular TCP. We examined experiments where we do not allow Nice's congestion window to fall below 1, i.e. (iii) is absent. In this case, when the number of background flows exceeds about 10, the latency of foreground flows begins to increase noticeably; the increase is about a factor of two when the number of background flows is 64. If (ii) is removed from Nice, its behavior is similar to TCP Vegas. In fact, Vegas already decreases its window size linearly when it detects a certain number of packets queued at the bottleneck router. If (i) is removed from Nice, it behaves similar to regular TCP. Increasing the aggressiveness of backoff

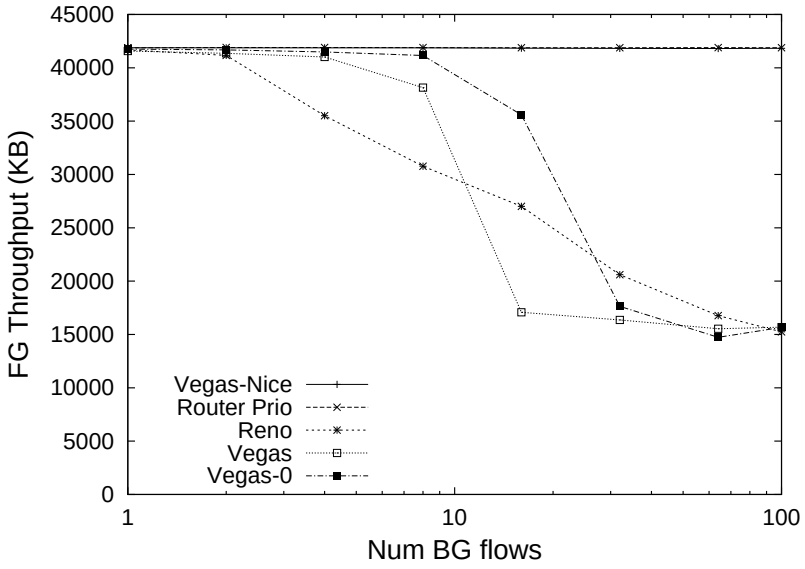



Figure 3.6: Number of BG flows vs FG throughput

without an early congestion detection mechanism like in GAIMD [184] or binomial congestion control [15] is not sufficient to provide the non-interfering property as the congestion signal, a packet drop, arrives too late.

**Experiment 3: Sensitivity to parameters** In this experiment we trace the effect of the threshold and trigger fraction parameters described in Section 3.1.2. Figure 3.7 shows the document transfer latencies as a function of the threshold for the same trace as above, with $S = 1$ and 16 background flows. As expected, as the threshold value increases, the interference caused by Nice increases until the protocol finally reverts to Vegas behavior as the threshold approaches 1. It is interesting to note that there is large range of threshold values yielding low interference, which suggests that its value need not be manually tuned for each network. As explained in section 3.2, in a well designed network, this phenomenon is to be expected from the expression for the bound obtained for interference in Theorem 1. We examine

the trigger fraction in the same way, and find little change in foreground latency as we vary this fraction from 0.1 to 0.9 as shown in Figure 3.8. This phenomenon is because of packets getting sent out in bursts and therefore observing similar round-trip delays.

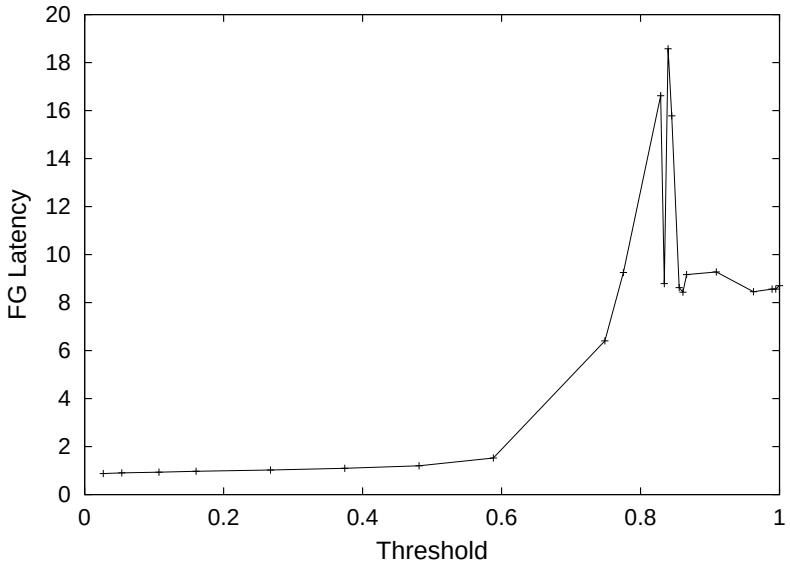



Figure 3.7: Threshold vs FG latency

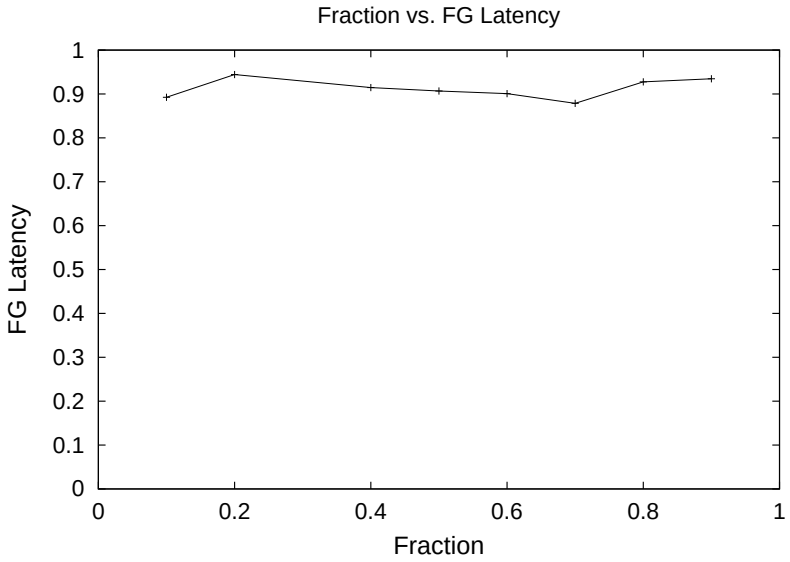



Figure 3.8: Fraction vs FG latency

**Experiment 4: Nice with RED queueing** We repeat experiments 1 and 2, but with routers performing RED queueing. The purpose of this experiment is to evaluate the performance of Nice if co-deployed with router-assisted congestion control mechanisms. The RED parameters are set as recommended in [75] with the "gentle" mode on. The minimum and maximum RED thresholds are set to one-third and two-third of the buffer size. Packets are probabilistically marked with ECN support from the senders. Figure 3.9 plots the foreground document transfer latency against the spare capacity with 16 background flows. Though Nice still performs as much as an order of magnitude better than other protocols, it causes up to a factor of 2 increase in document transfer latencies for large spare capacities. As figure 3.10 indicates, under RED, Nice closely approximates router prioritization regardless of the number of flows when the spare capacity is one, i.e. the demand workload consumes half of the network capacity.

The relatively poor performance of Nice under RED when spare capacities are large appears to reflect the sensitivity of Nice's interference $I$ to bottleneck queue length (Equation 1). Whereas Nice flows damage foreground flows when drop-tail queues are completely full, under RED, interference can begin when the bottleneck queue occupancy reaches RED's minimum threshold $min_{th}$. One solution may be to reduce Nice's $threshold$ parameter. The *Nice-0.03* lines in Figures 6 and 7 plot Nice's interference under RED when $threshold = 0.03$ instead of the default value of 0.10. Figure 3.11 shows the variation of interference with the reduced value of threshold and Figure 3.12 shows the corresponding reduction in the throughput obtained by background flows. Figure 3.11 shows that there is a small range of values of threshold for which Nice causes low interference. Intuitively, RED is similar to Nice in causing multiplicative backoff on a threshold queue build up. Nice's threshold thus needs to be much smaller than RED's to provide the effect of prioritized flows. Future work is needed to better understand Nice's interaction with RED queuing and develop

techniques to automatically adapt to any value of the RED threshold.

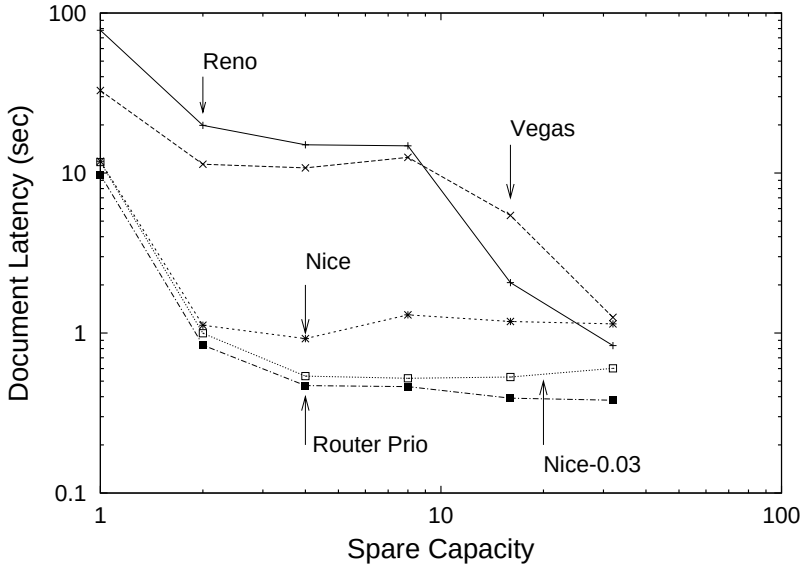



Figure 3.9: Spare capacity vs Latency with RED queueing

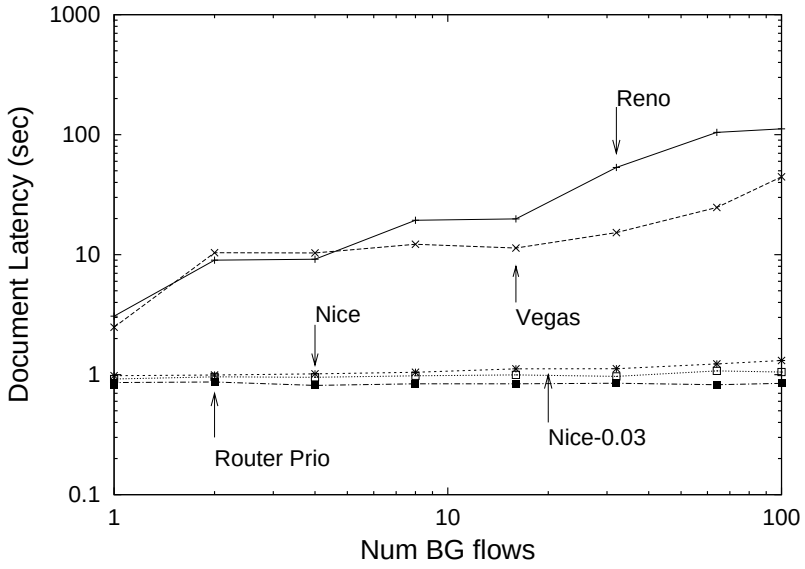



Figure 3.10: Number of BG flows vs Latency with RED queueing

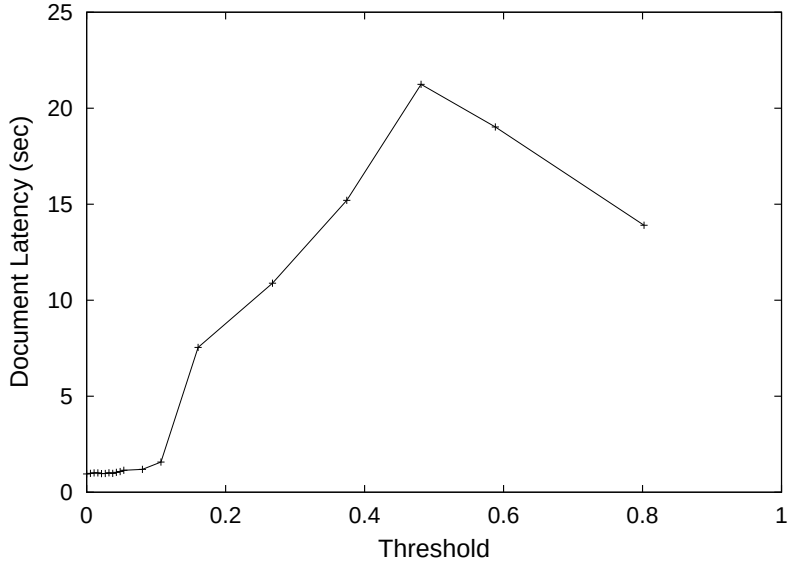



Figure 3.11: Threshold vs Latency with RED queueing

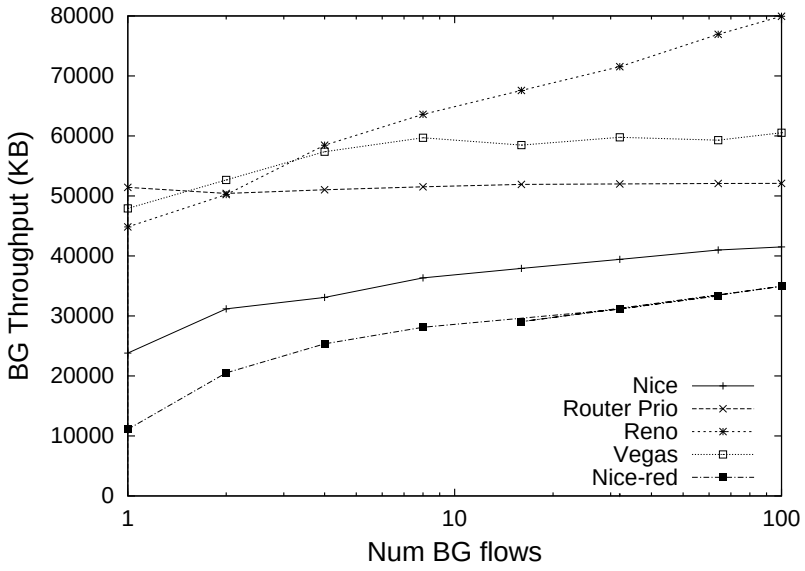



Figure 3.12: Number of BG flows vs BG throughput with RED queueing

**Experiment 5: Comparision with rate limiting** In this experiment we compare Nice to simple rate-limited Reno flows. The foreground traffic is again modeled by the Squid trace and the experiment performed is identical to experiment 1.

Fig 3.13 plots the average latency of foreground packets as a function of the spare capacity in the network. The various lines represent rate-limited background flows with the limits corresponding to a window size of 1,2,4 and 16. *It can be seen that even a flow with a rate limit of 1 inflicts slightly greater interference than Nice* This result is not surprising as Nice is equipped to reduce its window size below one when it deems necessary to minimize interference. All other flows with higher rates perform much worse and result in upto two orders of magnitude of increase in latency.

Next, we fix the capacity of the network to $S = 1$ (L twice the bandwidth needed by demand flows), and we vary the number of background flows. Figure 3.13 plots of the latency of foreground packets against the number of background flows. We observe that even flows limited to a window size of 1 inflict upto two orders of magnitude of increase in latency when there are 64 background flows present. Nice on the other hand is hardly distinguishable from the router prioritization line even for a 100 background flows (Figure 3.14). Figure 3.15 and 3.16 plots the number of bytes the background flows manage to transfer. We observe that a single Nice background flow gets more throughput than a flow rate limited to a window size of 8. This single Nice flow obtains about 10 times as much throughput as a flow rate-limited to a window of one but still causes lower intereference as was seen in the previous graph. With increasing number of flows, the rate-limited flows show a linear (X-axis is on a log-scale) increase in throughput while the throughput obtained by Nice increases much slower. However, all the rate-limited flows, sooner or later cross the router prioritization line, which means that they steal bandwidth from the foreground flows (figure 3.16). Nice on the other hand remains below the router

prioritization line always and gets between 60-80% of the spare bandwidth.

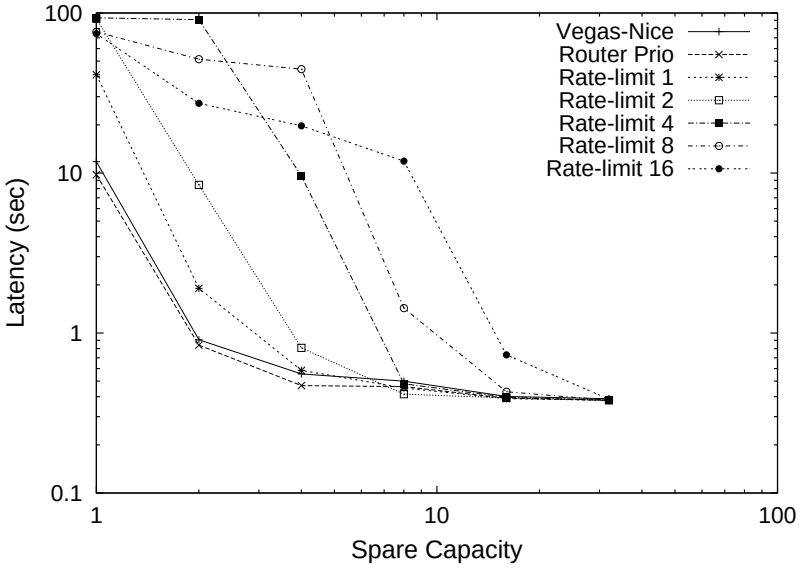



Figure 3.13: Spare capacity vs Latency

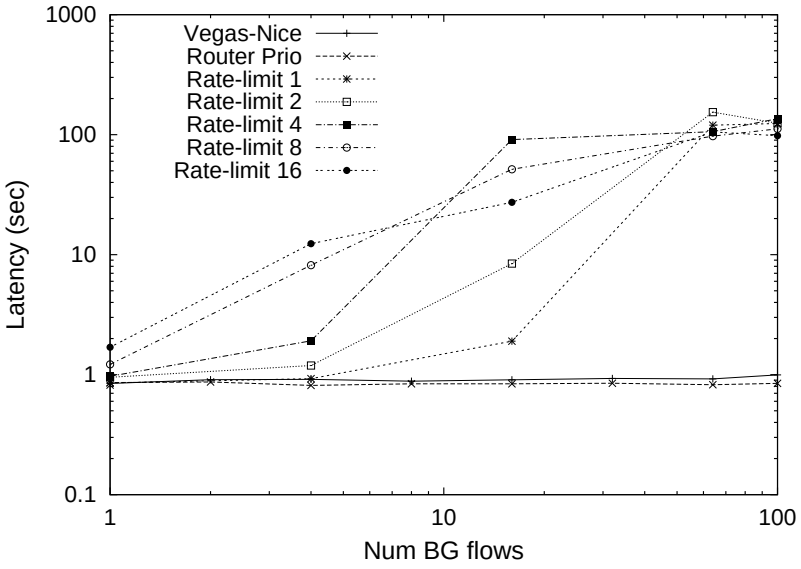



Figure 3.14: Number of BG flows vs Latency

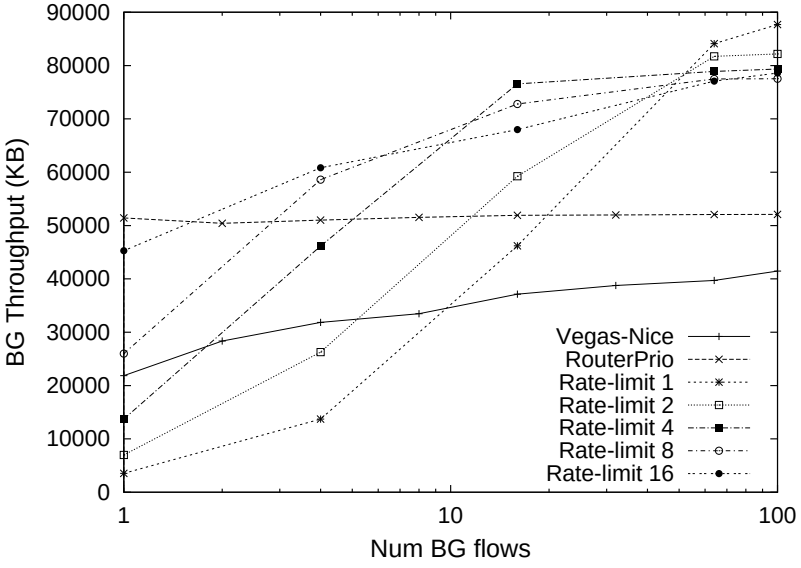



Figure 3.15: Number of BG flows vs BG throughput

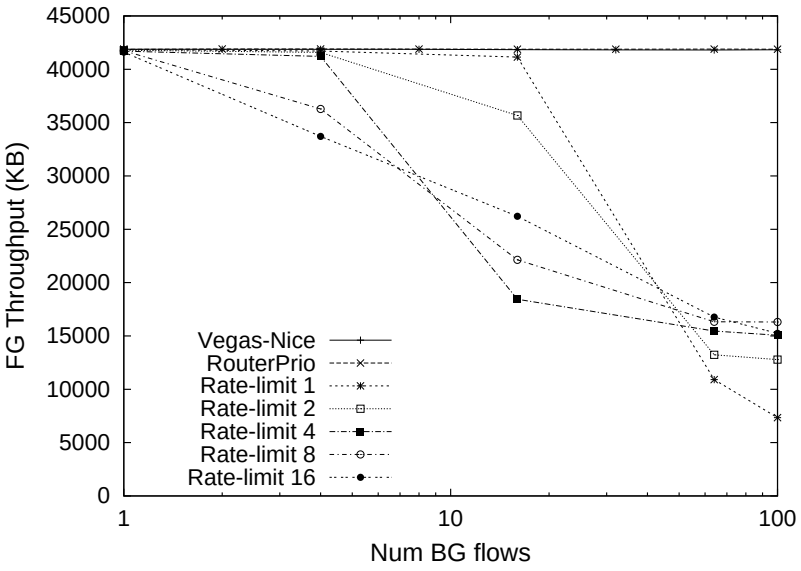



Figure 3.16: Number of BG flows vs FG throughput

**Experiment 6: Nice with random on/off UDP traffic** In this experiment we model the foreground traffic as a set of UDP sources transmitting in an on/off manner in accordance with a Pareto distribution. The burst time and idle time were each set to 250ms, and the value of the *shape* parameter set to 1.5 . The experiments performed were identical to the ones involving trace-based traffic *i.e.* spare capacity and the number of background flows were varied.

Fig 3.17 plots the average latency of foreground packets as a function of the spare capacity in the network. We observe that though the Nice flows cause less latency overhead than Reno or Vegas, the numbers are not as impressive as in the case when the foreground traffic was a trace following TCP. These numbers suggest that Nice is not well-suited to environments where the traffic is unpredictable. They also support our thesis that Nice works by using round-trip delay estimates in the current round to predict the state of the network in the next. As expected it doesn't work well when the traffic is unpredictable.

Next, we fix the capacity of the network to $S = 2$ (L has four time the bandwidth needed by demand flows), and we vary the number of background flows. Fig 3.18 plots the latency of foreground packets against the number of background flows. We observe that though Nice outperforms Reno and Vegas, it doesn't match router prioritization as closely. However, Nice continues to show relatively graceful degradation with the number of background flows because of it's ability to decrease its window size below one. As in the case of TCP flows, Figure 3.20 shows that Nice hardly affects the throughput obtained by the on/off UDP cross traffic. Figure 3.19 plots the number of bytes the background flows manage to transfer. We observe that a single Nice flow obtains about 70% of the spare bandwidth available under router prioritization; this background throughput improves with increasing number of background flows but remains below router prioritization. Thus, Nice reaps a significant fraction of the spare capacity (at the cost of increased interference) when

the foreground traffic is unpredicatble. Nice has been designed to cause low interference with well-behaved TCP flows. Hence, the relatively unimpressive performance of Nice in the midst of random on/off UDP traffic is not surprising.

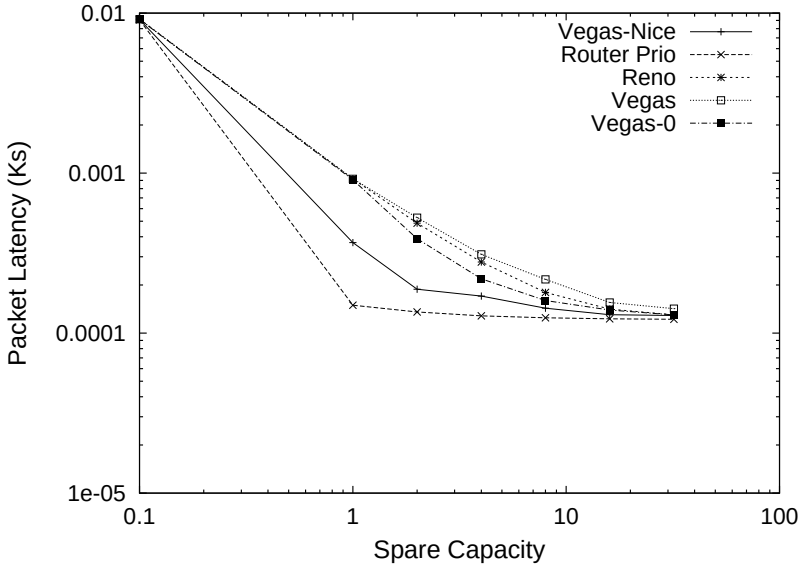



Figure 3.17: Spare capacity vs Latency with on/off UDP cross traffic

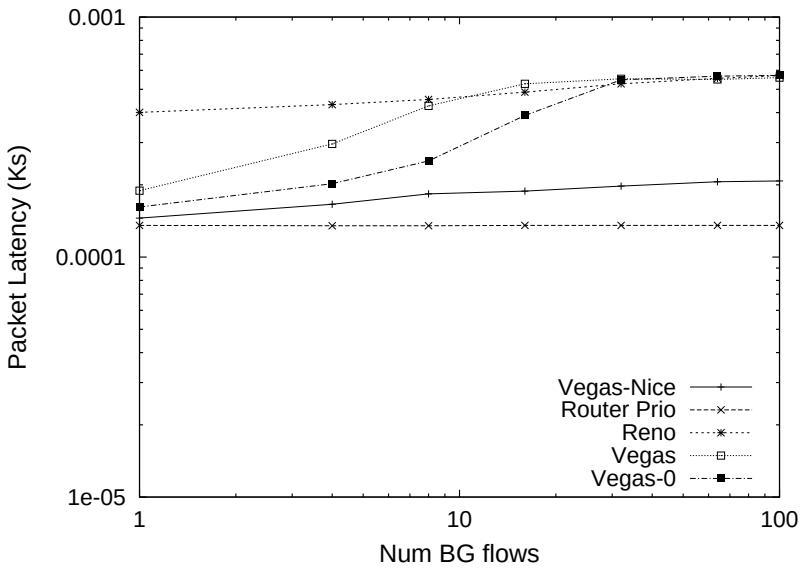



Figure 3.18: Number of BG flows vs Latency with on/off UDP cross traffic

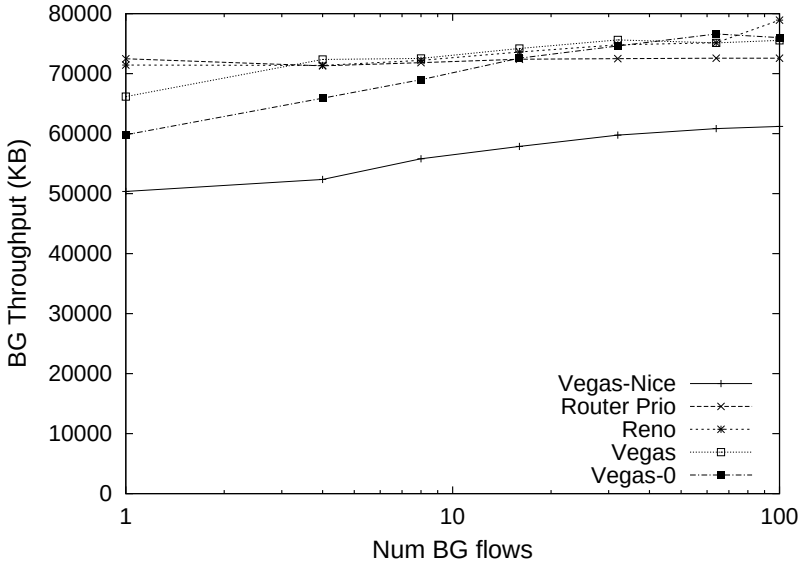



Figure 3.19: Number of BG flows vs BG throughput with on/off UDP cross traffic

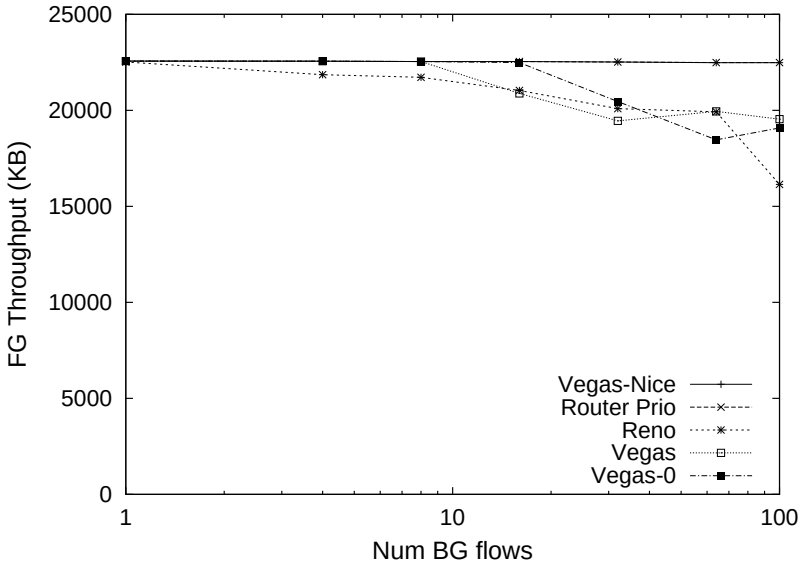



Figure 3.20: Number of BG flows vs FG throughput with on/off UDP cross traffic

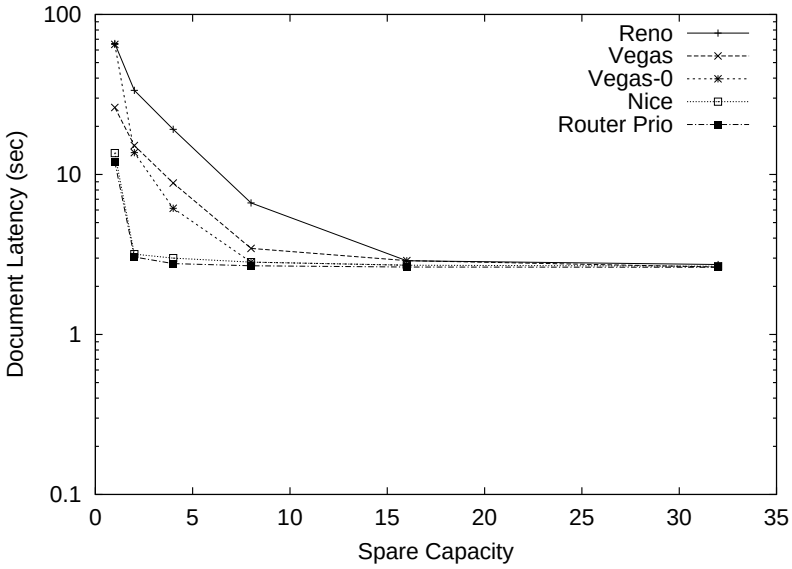



Figure 3.21: Spare capacity vs Latency with a Nice's RTT four times that of foreground traffic

**Effect of RTT Bias** TCP is known to be unfair biased towards flows with shorter round-trip times that share a common bottleneck; flows receive throughout inversely proportional to their RTT. In this experiment we study the effect of dissimilar RTTs on Nice's non-interference property. In particular, we seek to understand if Nice flows with short RTTs end up inflicting an unacceptably high amount of interference on TCP flows with much higher RTTs. To this end, we repeat experiments 1 and 2 with foreground traffic having an RTT four times higher than the background flows. The queue capacity was set to bandwidth-delay product where the delay used was the average of the two RTT values. Figures 3.21-3.24 show the results of this experiment. As can be seen, Nice continues to maintain its strong non-interference property, while other candidate background protocols cause more than an order of magnitude of increase in document transfer times.

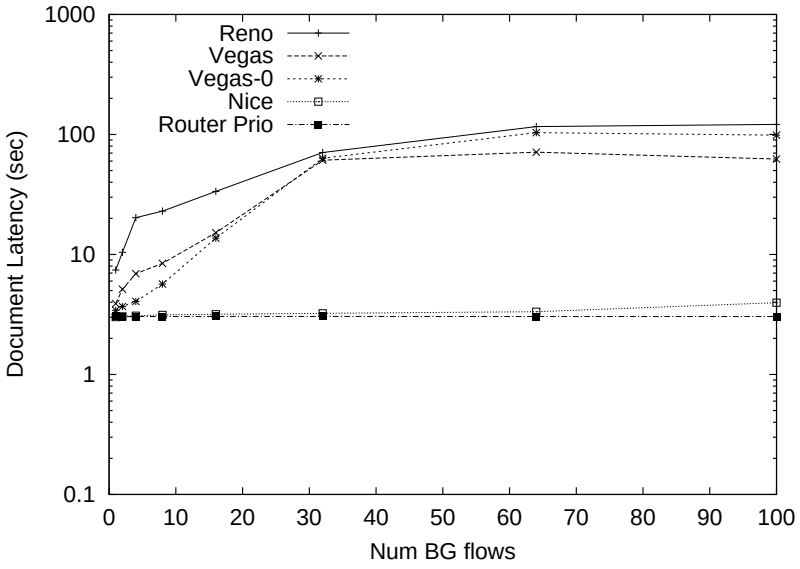



Figure 3.22: Number of BG flows vs Latency with Nice's RTT four times that of foreground traffic

We note that if the queue capacity were left unchanged in this experiment, i.e. if it were set to the bandwidth-delay product using the lower of the two RTTs, Nice's interference increases slightly causing up to a 30-40% increase in average document transfer latency beyond 32 Nice flows given a spare capacity of 1 at the bottleneck router. This increased interference is expected as Nice flows do not have sufficient time to back off before packet drops occue if queue capacities are small. Our theoretical analysis also supports this observation.

**Reverse Path Traffic** In the above experiments, all of the traffic flowed in only one direction. Traffic in the reverse direction along the same path can affect congestion control in the forward direction. For example, if acknowledgements are delayed because of traffic on the reverse path, then the throughput achieved by a flow on the forward path can get reduced. Traffic on the reverse path can affect the robustness

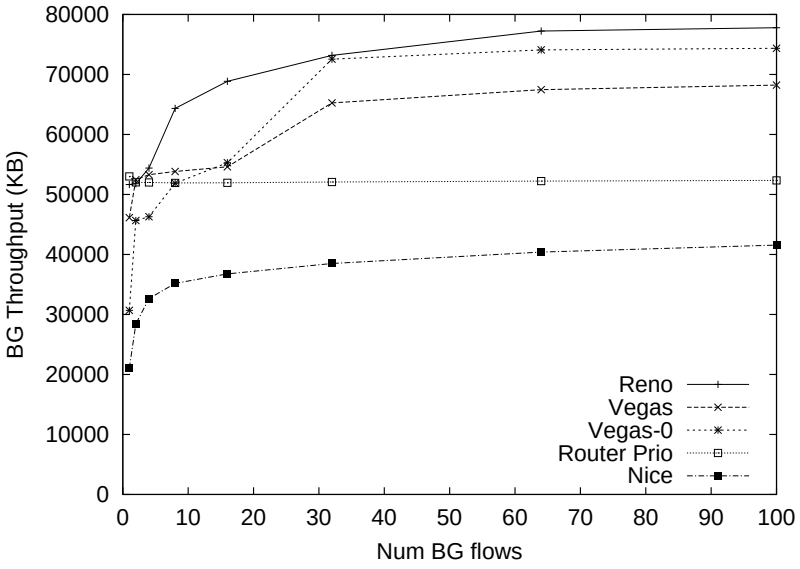



Figure 3.23: Number of BG flows vs BG throughput with Nice's RTT four times that of foreground traffic

of delay-based congestion control protocolss [147]. For example, reverse path traffic can add noise to RTT measurements done by a delay-based congestion controlled flow. If the sending rate of such a flow is small, reverse path traffic, and cross traffic in general, can lead to undersampling of network state by the flow.

To determine the effect of reverse-path traffic on Nice's interference property, we introduce two long-lived FTP transfers in the reverse path and repeat experiments 1 and 2. Figures 3.25-3.28 show the corresponding results. We observe that though Nice's interference continues to be low, it is somewhat higher than in experiments 1 and 2 without reverse path traffic. For example, 100 background flows cause a 22% increase in average document transfer latency given a spare capacity (with respect to the forward path traffic as before) of 1 as opposed to a 14% increase when there is no reverse path traffic. This increased interference is because of the small probing overhead of TCP Nice in an environment with very little spare capac-

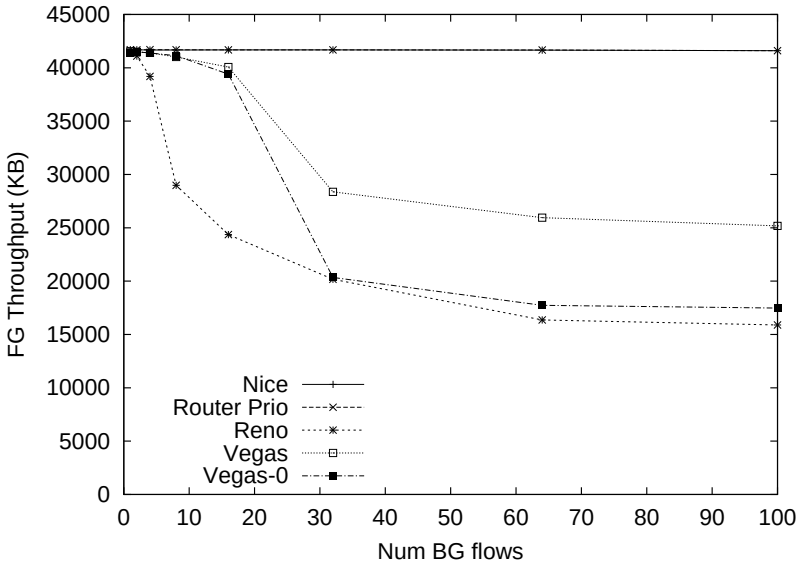



Figure 3.24: Number of BG flows vs FG throughput with Nice's RTT four times that of foreground traffic

ity. Unlike the traffic on the forward path that is intermittent in nature, the FTP transfers always attempt to use up all of the capacity at the bottleneck router leaving little spare capacity to be reaped by background flows as shown in Figure 3.28. We do note, however, that Nice's interference does not increase significantly with increasing number of background flows and continues to be an order of magnitude less than other candidate background protocols when the number of background flows is large.

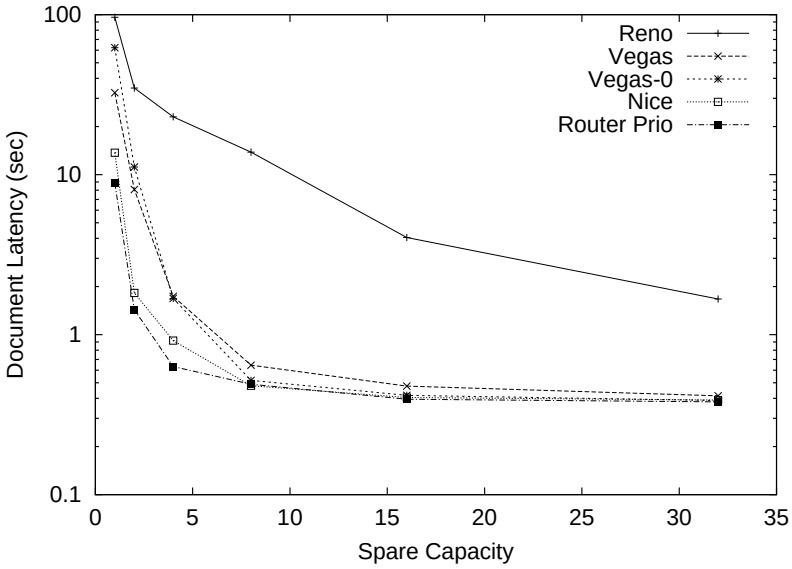



Figure 3.25: Spare capacity vs Latency with traffic on the reverse path

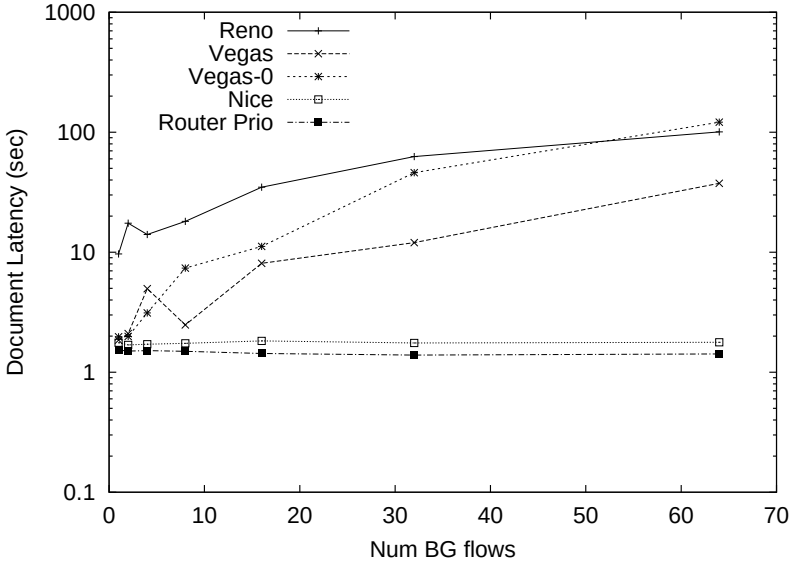



Figure 3.26: Number of BG flows vs Latency with traffic on the reverse path

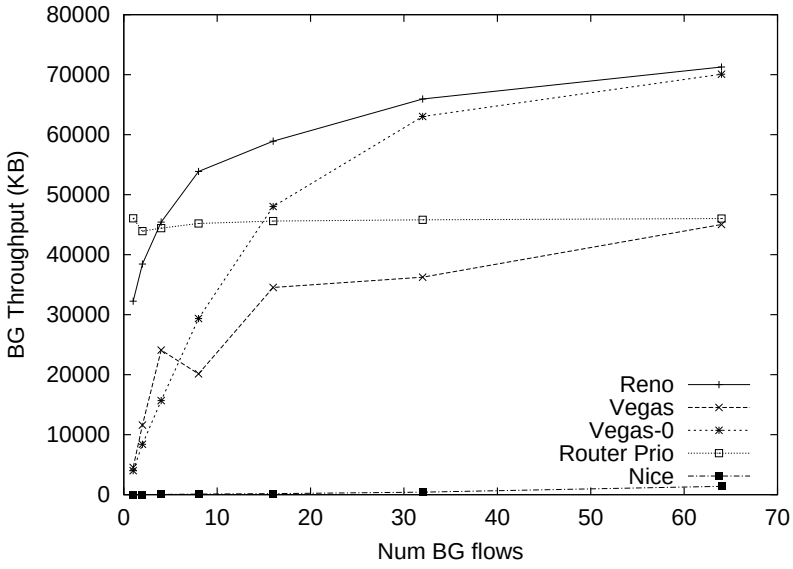



Figure 3.27: Number of BG flows vs BG throughput with traffic on the reverse path

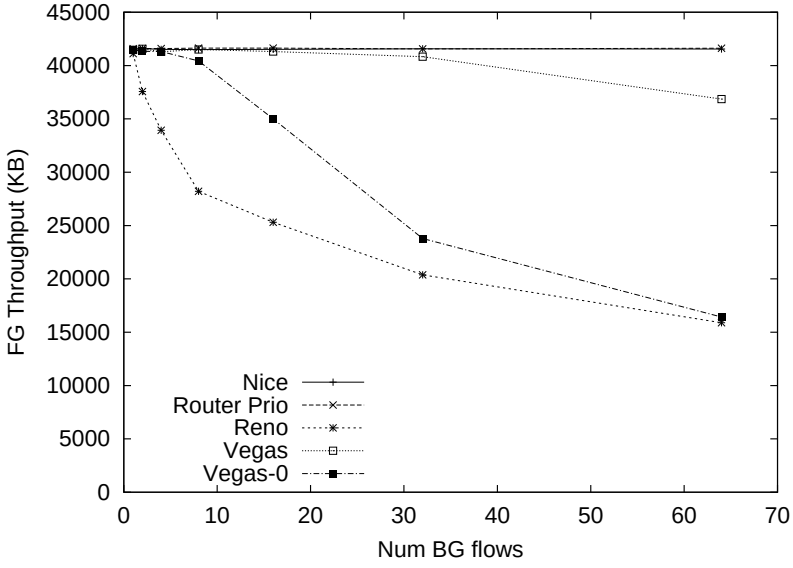



Figure 3.28: Number of BG flows vs FG throughput with traffic on the reverse path

## 3.4 Internet Microbenchmarks

In this section we evaluate our Nice implementation over a variety of Internet links. We seek to answer three questions. First, in a less controlled environment than our NS simulations, does Nice still avoid interference? Second, are there enough reasonably long periods of spare capacity on real links for Nice to reap reasonable throughput? Third, are any such periods of spare capacity spread throughout the day, or is the usefulness of background transfers restricted to nights and weekends?

Our experiments suggest that Nice works for a range of networks, including a modem, a cable modem, a transatlantic link, and a fast WAN. In particular, on these networks it appears that Nice avoids interfering with other flows and that it can achieve throughput that are significant fractions of the throughput that would be achieved by Reno throughout the day.

### 3.4.1 Methodology

Our measurement client program connects to a measurement server program at exponentially-distributed random intervals. At each connection time, the client chooses one of six actions: Reno/NULL, Nice/NULL, Reno/Reno, Reno/Nice, Reno/Reno8, Reno/Nice8 (We also tested standard Vegas in place of Reno for the large-transfer experiments and find that standard Vegas behaves essentially like Reno). Each action consists of a "primary transfer" (denoted by the term left of the /) and zero or more "secondary transfers" (denoted by the term right of the /). Reno terms indicate flows using standard TCP-Reno congestion control. Nice terms indicate flows using Nice congestion control. For secondary transfers, NULL indicates actions that initiate no secondary transfers to compete with the primary transfer, and 8 indicates actions that initiate 8 (rather than the default 1) secondary transfers. The transfers are of large files whose sizes are chosen to require approximately 10 seconds for a single Reno flow to compete on the network under study.

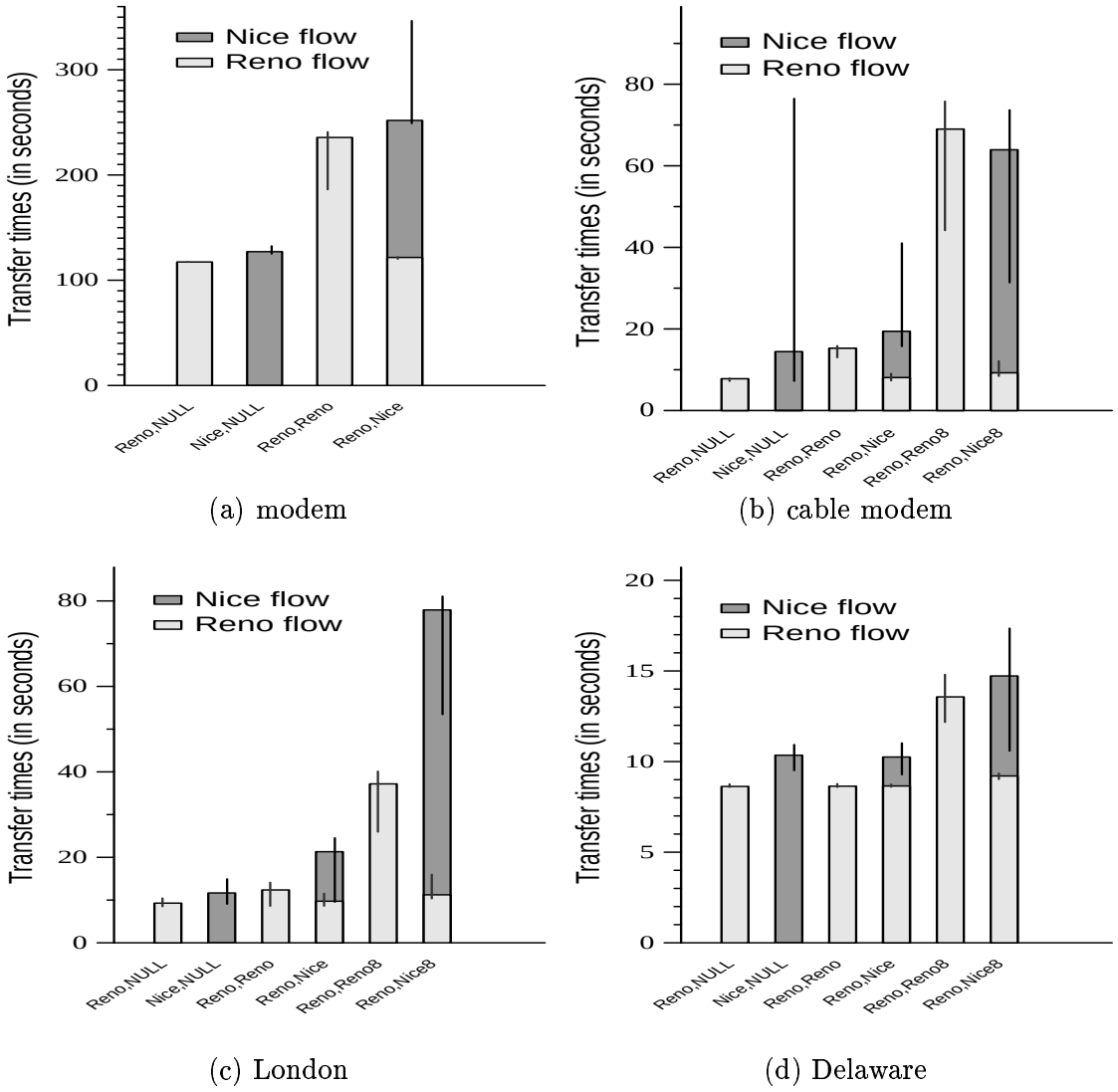
We position a server that supports Nice at UT Austin. We position clients (1) in Austin connected to the Internet via a University of Texas 56.6K dial in modem bank (*modem*), (2) in Austin connected via a commercial ISP cable modem (*cable modem*), (3) in a commercial hosting center in London, England connected to multiple backbones including an OC12 and an OC3 to New York (*London*), and (4) at the University of Delaware, which connects to UT via an Abilene OC3 (*Delaware*). All machines run Linux. The server is a 450MHz Pentium II with 256MB of memory. The clients range from 450-1000MHz and all have at least 256MB of memory. The experiment ran from Saturday May 11 2002 to Wednesday May 15 2002; we gathered approximately 50 probes per client/workload pair.

### 3.4.2 Results

Figure 3.29 summarizes the results of our large-transfer experiments. On each of the networks, the throughput of Nice/NULL is a significant fraction of that of Reno/NULL, suggesting that periods of spare capacity are often long enough for Nice to detect and make use of them. Second, we note that during Reno/Nice and Reno/Nice8 actions, the primary (Reno) flow achieves similar throughput to the throughput seen during the control Reno/NULL sessions. In particular, on a modem network, when Reno flows compete with a single Nice flow, they receive on average 97% of the average bandwidth they receive when there is no competing Nice flow. On a cable modem network, when Reno flows compete with eight Nice flows, they receive 97% of the bandwidth they would receive alone. Conversely, Reno/Reno and Reno/Reno8 show the expected fair sharing of bandwidth among Reno flows, which reduces the bandwidth achieved by the primary flow.

Figures 3.4.2(a),3.4.2(b), and 3.4.2(c) show the hourly average bandwidth achieved by the primary flow for the different combinations listed above. Our hypothesis is that Nice can achieve useful amounts of throughput throughout the day,

and the data appear to support this statement.



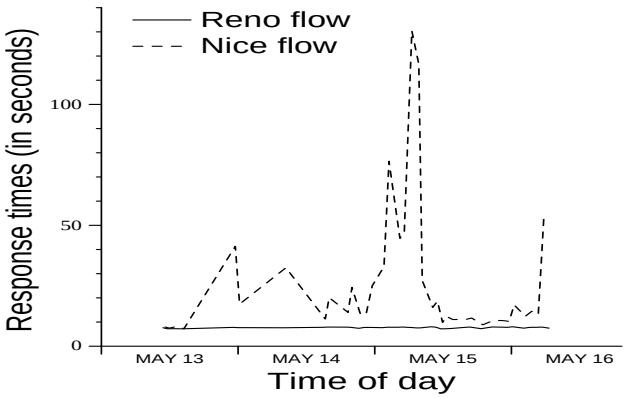

(a) modem

(b) cable modem

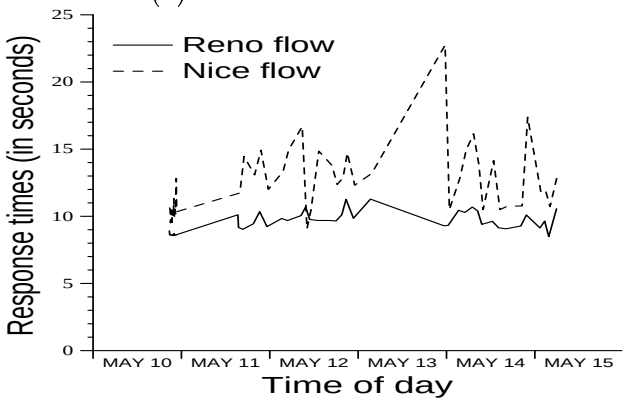
(c) London

(d) Delaware

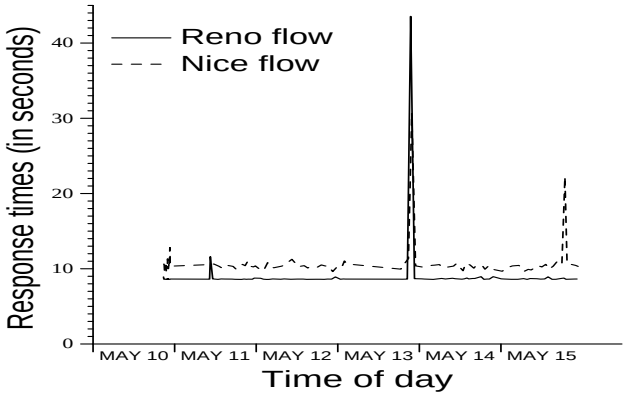
Figure 3.29: Large flow transfer performance. Each bar represents the average transfer time observed for the specified combination of primary/secondary transfers. Empty bars represent the average time for a Reno flow. Solid bars represent the average time for a Nice flow. The narrow lines depict the minimum and maximum values observed during multiple runs of each combination.




(a) Cable modem network




(b) Austin to London




(c) U.T. Austin to U. Delaware

Figure 3.30: Large flow transfer performance over time

## 3.5 Case Study Applications

### 3.5.1 HTTP Prefetching

Many studies have published promising results that suggest that prefetching (or pushing) content could significantly improve web cache hit rates by reducing compulsory and consistency misses [50, 64, 86, 89, 114, 140, 174].

Typically, prefetching algorithms are tuned with a *threshold* parameter to balance the potential benefits of prefetching data against the bandwidth costs of fetching it and the storage cost of keeping it until its next use. An object is prefetched if the estimated probability that the object will be referenced before it is modified exceeds the threshold. Extending Gray and Shenoy's analysis of demand caching [82], Chandra calculates reasonable thresholds given network costs, disk costs, and human waiting time values and concludes that most algorithms in the literature have been far too conservative in setting their thresholds [37]. Furthermore, the 80-100% per year improvements in network [37, 139] and disk [55] capacity/cost mean that a value that is correct today may be off by an order of magnitude in 3-4 years.

In this case study, we build a prefetching protocol similar to the one proposed by Padmanabhan and Mogul [140]: when serving requests, servers piggy back lists of suggested objects in a new HTTP reply header. Clients receiving a prediction list discard old predictions and then issue prefetch requests of objects from the new list. This division of labor allows servers to use global information and application-specific knowledge to predict access patterns, and it allows clients to filter requests through their caches to avoid repeatedly fetching an object.

To evaluate prefetching performance, we implement a standalone client that reads a trace of HTTP requests, simulates a local cache, and issues demand and prefetch requests. Our client is written in Java and pipelines requests across HTTTP/1.1 persistent connections [72]. To ensure that demand and prefetch requests use separate TCP connections, our server directs prefetch requests to a different port than

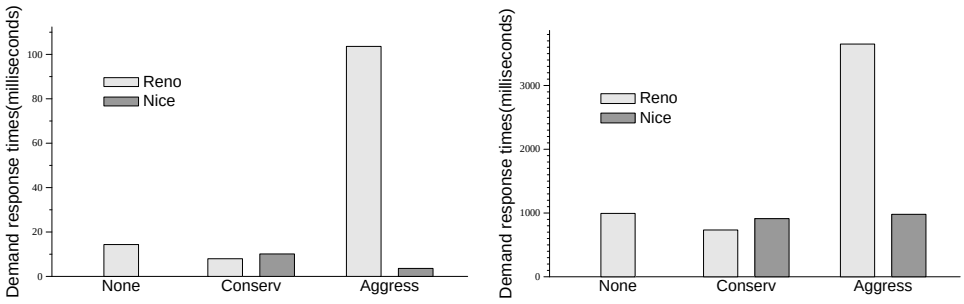
demand requests. The disadvantage of this approach is that it does not fit with the standard HTTP caching model. We discuss how to deploy such a protocol without modifying HTTP in a separate study [109].

We use Squid proxy traces from 9 regional proxies collected during January 2001 [179]. We study network interference near the server by examining subsets of the trace corresponding to a popular groups of related servers – *cnn* (e.g., cnn.com, www.cnn.com, cnnfn.com, etc.). This study compares relative performance for different resource management algorithms for a given set of prefetching algorithms. It does not try to identify optimal prefetching algorithms; nor does it attempt to precisely quantify the absolute improvements available from prefetching.

We use a simple prediction by partial matching algorithm [47] PPM-$n/w$ that uses a client's $n$ most recent requests to the server group for non-image data to predict cachable (i.e., non-dynamically-generated) URLs that will appear during a subsequent window that ends after the $w$'th non-image request to the server group. We use two variations of our PPM-$n/w$ algorithm. The *conservative* variation uses parameters similar to those found in the literature for HTTP prefetching. It uses $n = 2$, $w = 5$ and sets the prefetch threshold to 0.25 [64]. To prevent prefetch requests from interfering with demand requests, it pauses 1 second after a demand reply is received before issuing requests. The *aggressive* variation uses $n = 2$, $w = 10$, and truncates prefetch proposal lists with a threshold probability of 0.00001. It issues prefetches immediately after receiving them.

We use 2 client machines connected to a server machine via a cable modem. On each client machine, we run 8 virtual clients, each with a separate cache and separate HTTP/1.1 demand and prefetch connections to the server. In order for the demand traffic to consume about 10% of the cable modem bandwidth, we select the 6 busiest hours from the 30-Jan-2001 trace and divide trace clients from each hour randomly across 4 of the virtual clients. In each of our seven trials, all the

16 virtual clients run the same prefetching algorithm: *none*, *conservative-Reno*, *aggressive-Reno*, *conservative-Nice*, *aggressive-Nice*.




(a) cable modem (b) dial-up modem

Figure 3.31: Average demand transfer time for prefetching for the cnn server-group.

Figure 3.31(a) shows the average demand response times perceived by the clients. We note that when clients do conservative prefetching using either protocol – Nice or Reno – the latency reductions are comparable. However, when they start aggressively prefetching using Reno, the latency blows up by an order of magnitude. Clients using aggressive Nice prefetching, however, continue to see further latency reductions. The figure shows that Nice is effective in using spare bandwidth for prefetching without affecting the demand requests.

Figure 3.31(b) represents the effect of prefetching over a modem (the setup is same as above except with the cable modem replaced by a modem), an environment where the amount of spare bandwidth available is minimal. This figure shows that while the Reno and Nice protocols are comparable in benefits when doing conservative prefetching, aggressive prefetching using Reno hurts the clients significantly by increasing the latencies three-fold. Nice on the other hand, does not worsen the latency even though it does not gain much.

We conclude that Nice simplifies the design of prefetching applications. Applications can aggressively prefetch data that might be accessed in the future. Nice

prevents interference if the network does not have spare bandwidth and improves application performance if it does.

### 3.5.2 Tivoli Data Exchange

We study a simplified version of the Tivoli Data Exchange [67] system for replicating data across large numbers of hosts. This system distributes data and programs across thousands of client machines using a hierarchy of replication servers. Both non-interference and good throughput are important metrics. In particular, these data transfers should not interfere with interactive use of target machines. And because transfers may be large, may be time critical, and must go to a large number of clients using a modest number of simultaneous connections, each data transfer should complete as quickly as possible. The system currently uses two parameters at each replication server to tune the balance between non-interference and throughput. One parameter throttles the maximum rate that the server will send a single client; the other throttles the maximum total rate across all clients.

Choosing these rate limiting parameters requires some knowledge of network topology and may have to choose between overwhelming slow clients and slowing fast clients (e.g., distributing a 300MB Office application suite would nearly a day if throttled to use less than half a 56.6Kb/s modem). One could imagine a more complex system that allows the maximum bandwidth to be specified on a per-client basis, but such a system would be complex to configure and maintain.

Nice can provide an attractive self-tuning abstraction. Using it, a sender can just send at the maximum speed allowed by the connection. We report preliminary results using a standalone server and client. The server and clients are the same as in the Internet measurements described in Section 3.4. We initiate large transfers from the server and during that transfer measure the ping round trip time between the client and the server. When running Reno, we vary the client throttle parameter and

leave the total server bandwidth limit to an effectively infinite value. When running Nice, we set both the client and server bandwidth limits to effectively infinite values.

Figure 3.32 shows a plot of ping latencies (representative of interference) as a function of the completion time of transfers to clients over different networks. With Reno, completion times decrease with increasing throttle rates but increase ping latencies as well. Furthermore, the optimal rates vary widely across different networks. However Nice picks sending rates for each connection without the need for manual tuning that achieve minimum transfer times while maintaining acceptable ping latencies in all cases.

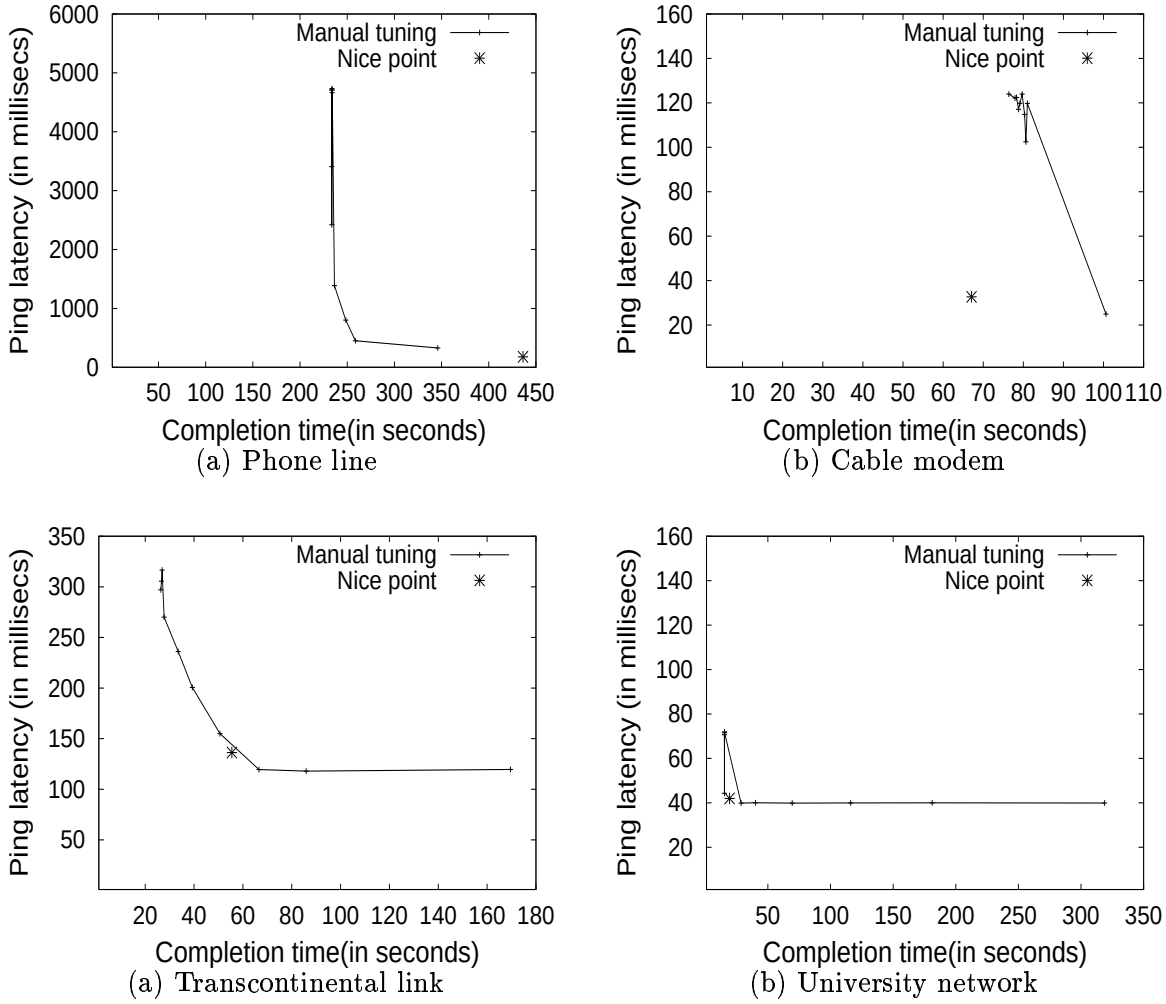



Figure 3.32: Each continuous line represents completion times and corresponding ping latencies with varying send rates. The single point is the send rate chosen by Nice.

## 3.6 Related work

TCP congestion control has seen an enormous body of work since Jacobson's seminal paper on the topic [103]. This work seeks to maximize utilization of network capacity, to share the network fairly among flows, and to prevent pathological scenarios like congestion collapse. In contrast, our primary goal is to ensure minimal interference with regular network traffic; though high utilization is important, it is a distinctly subordinate goal in our algorithm. Our algorithm is always less aggressive than AIMD TCP: it reacts the same way to losses and in addition, it reacts to increasing delays. Therefore, the work to ensure network stability under AIMD TCP applies to Nice as well.

The work on TCP-LP, a transport protocol for low priority tranfers, is closest to our work on TCP Nice. Though the core features of Nice and TCP-LP are similar, the context of our work on Nice is broader with a focus on designing large-scale replicated systems. Our case studies using Nice for building a data distribution system and a Web prefetching system demonstrate end-to-end benefits in real applications. A technical difference between Nice and TCP-LP is that in Nice, being based on TCP Vegas, the window size does not fluctuate under unchanging network conditions whereas in TCP-LP, similar to standard TCP, the window continues to oscillate in a small range.

The GAIMD [184] and binomial [15] frameworks provide generalized families of AIMD congestion control algorithms to allow protocols to trade smoothness for responsiveness in a TCP-friendly manner. The parameters can also be tuned to make a protocol less aggressive than TCP. We considered using these frameworks for constructing a background flow algorithm, but we were unable to develop the types of strong non-interference guarantees we seek using these frameworks. One area for future work is developing similar generalizations of Nice in order to allow different background flows to be more or less aggressive compared to one another

while all remain completely timid with respect to competing foreground flows.

Prioritizing packet flows would be easier with router support. As noted in Section 3.3, router prioritization queues such as those proposed for DiffServ [21] service differentiation architectures are capable of completely isolating foreground flows from background flows while allowing background flows to consume nearly the entire available spare bandwidth. Unfortunately, these solutions are of limited use for someone trying to deploy a background replication service today because few applications are deployed solely in environments where router prioritization is installed or activated. A key conclusion of this study is that an end-to-end strategy need not rely on router support to make use of available network bandwidth without interfering with foreground flows.

RFC 3662 [22] describes a lower than best-effort per-domain behavior for Differentiated Services that can be used for background transfers or transfers of a "low value" for which delivery is optional. The intended application of this proposal is similar in spirit to that of TCP Nice. TCP Nice provides lower than best-effort service without modifying routers, whereas proposals based on Differentiated Services require modifications to routers.

Applications can limit the network interference they cause in various ways:

(a) *Coarse-grain scheduling*: Background transfers can be scheduled during hours where there is little foreground traffic. Studies [66, 128] show that prefetching data during off-peak hours can reduce latency and peak bandwidth usage.

(b) *Rate limiting*: Spring et al. [163] discuss prioritizing flows by controlling the receive window sizes of clients. Crovella et al. [50] propose a combination of window-based rate control and pacing to spread out prefetched traffic to limit interference. They show that such shaping of traffic leads to less bursty traffic and smaller queue lengths.

(c) *Application tuning*: Applications can limit the amount of data they send by

varying application-level parameters. For example, many prefetching algorithms estimate the probability that an object will be referenced and only prefetch that object if its probability exceeds some threshold [64, 86, 140, 174].

It is not clear how an engineer should go about setting such application-specific parameters. We believe that self-tuning support for background transfers has at least three advantages over existing application-level approaches. Nice operates over fine time-scales, so it can provide lower interference (by reacting to spikes in load) as well as higher average throughput (by using a large fraction of spare bandwidth) than static hand-tuned parameters. This property reduces the risk and increases the benefits available to background transfers while simplifying application design. Our experiments also demonstrate that Nice provides useful bandwidth throughout the day in many environments.

Existing transport layer solutions can be used to tackle the problem of self-interference between a single sender/receiver's flows. The congestion manager CM [8] provides an interface between the transport and the application layers to share information across connections and for handling applications using different transport protocols. Microsoft XP's Background Intelligent Transfer Service (BITS) provides support for transfers of lower priority to minimize interference with the user's interactive sessions by using a rate throttling approach. In contrast to these approaches, Nice handles both self- as well as cross-interference by modifying the sender side alone.

## 3.7 Discussion

**Fairness in TCP Nice** We do not explicitly quantify fairness *among* Nice flows in our experiments. However, we remark that we expect TCP Nice to have fairness properties similar to regular TCP. The AIMD property lets flows sharing the same bottleneck router and same round-trip times to converge to an equal distribution of

the bottleneck bandwidth amongst them as shown in by Chiu and Jain [45]. Since TCP Nice essentially behaves like TCP with a reduced queue size, it continues to uphold TCP's fairness property and the analysis by Chiu and Jain can be straightforwardly extended to show the same. TCP itself has a round-trip time bias in the way it distributes capacity across flows with different round-trip times. At a bottleneck router, flows with longer round-trip times receive a proportionately smaller fraction of throughput. TCP Nice will also exhibit such a bias in favour of flows with shorter round-trip times.

**Highly Congested Networks** In practice, TCP Nice can quickly obtain reasonable values of $minRTT$ and $maxRTT$ essential for the non-interference property. However, it is conceivable that in an extremely congested network, a Nice flow never observes the minimum round-trip time. A high estimate of $minRTT$ can inflict interference on regular TCP flows in an already congested network. TCP Nice may be augmented to use loss rates as well as a signal of congestion, in addition to increasing round-trip times, to handle situations of extreme congestion. On observing a high loss rate, a Nice flow can determine that its estimate of network conditions based on round-trip time measurements is inaccurate and accordingly take more aggressive backoff measures or stop transmitting packets altogether for some time.

**Multiple Levels of Priority** TCP Nice effectively provides a two-level prioritization of network services. Can the design features of Nice be extended to incorporate multiple levels of priority? Theoretically, if routers are equipped with very large buffer capacities, then TCP Nice could use different values of the threshold parameter that are reasonably spaced apart to obtain multiple levels of priority. A large buffer capacity at a bottleneck router gives each one of the different kinds of Nice flows sufficient time to back off without interfering with Nice flows at the next priority level. However, in practice, and as also affirmed by preliminary simulation

experiments, the non-interference property does not scale well to multiple levels of priority for practical values of buffer capacities. Providing multiple levels of priority in an end-to-end manner so that each level is non-interfering with respect to higher levels is an open problem and an avenue for future work.

## 3.8 Conclusions

In this chapter, we presented an end-to-end congestion control algorithm optimized to support background transfers. Surprisingly, an end-to-end protocol can nearly approximate the ideal router-prioritization strategy by (i) almost eliminating interference with demand flows and (ii) reaping significant fractions of available spare network bandwidth.

Our Internet experiments suggest that there is a significant amount of spare capacity on a wide variety of Internet links. Nice provides a mechanism to improve application performance by harnessing this capacity in a non-interfering manner. Our case studies demonstrate that Nice can simplify application design by eliminating the need to hand-tune parameters to balance utilization and interference. Inspired by the results in this paper, we have built a self-tuning prefetching system [109] based on Nice that avoids interference at the server and in the network, and is deployable with simple modifications to a web server.

One application of Nice is to support massive replication of data and services, where spare resources (e.g. bandwidth, disk space, and processor cycles) are consumed to help humans be more productive. Massive replication systems should be designed as if bandwidth were essentially free. TCP Nice provides a reasonable approximation of such an abstraction.

# Chapter 4

# Mars: A Self-tuning Replication Architecture

In this section, we present Mars, an architecture for constructing self-tuning large-scale replication systems. We motivate the need for such a solution by illustrating the limitations and risks of alternative approaches for speculative replication, based on manual tuning of system parameters, that are employed by engineers today. In particular, we perform a case study of this problem in the context of Web prefetching systems and show that the manual-tuning or *threshold-based* approach is complex, inefficient, and exposes systems to the risk of overload. We then show how Mars' architecture can be instantiated to construct a deployable self-tuning Web prefetching system, NPS, that is simple, efficient, and safe.

## 4.1 Threshold-based Speculative Replication

Manually-tuned thresholds in speculative replication systems attempt to balance issues of policy as well as mechanism using magic numbers. For instance, some Web prefetching systems [64, 102] prefetch a file if its value, i.e. the probability of

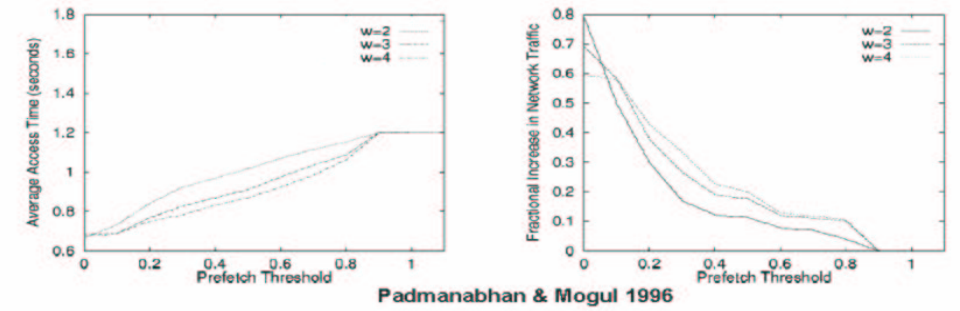



Figure 4.1: Response Time and Fractional Increase in Network Traffic as a function of the Prefetch Threshold

access before it expires, exceeds a certain threshold. The mechanism challenges that prefetching systems seek to address are those of interference at various points in the system, effective utilization of available resources, and maintaining robustness to the risk of system overload. The policy challenges that prefetching systems seek to address include optimizing response time, availability, and freshness of content while respecting bandwidth, storage and computing constraints at the server, network, and client. The threshold-based approach attempts to address both policy and mechanism challenges with a single number that determines the prefetch-worth of files.

One of the seminal papers that introduced the idea of predictive prefetching for the Web [140] by Padmanabhan et al. suggests such a threshold-based approach. They use a history-based Markov model to predict what files will be accessed next, based on the knowledge of the previous few files requested. A file is prefetched if its probability of access exceeds a certain threshold that is a fixed value. The two plots in Figure 4.1 taken from that paper show the average response time and the

fractional increase in network traffic as a function of the prefetch threshold used. As expected, increasing the aggressiveness of prefetching (by using smaller values of the threshold) improves response time and increases the overhead of network traffic, and vice-versa. On a cursory glance, it appears natural to tune the value of the threshold to balance the cost (network bandwidth) against the benefit (response time). Values in the vicinity of 0.25 are common [64, 140, 102] as they correspond to the "knee" of the cost curve. Such an approach, or variations thereof, have been widely suggested and shown to work well in simplistic laboratory simulations.

However, the threshold-based approach has some problems. The knee of the cost curve in the Padmanabhan study is strictly a function of the particular workload they considered. Moreover, the real hidden cost is that of interference and the associated risk of overload, not just additional bandwidth consumption. Attempting to use thresholds in the above manner presents the following problems :

**Problem 1** : It is unclear what choice of threshold will work correctly for a given scenario. The benefits, including improvements in latency and availability, and costs, including interference, bandwidth, storage, computing, and risk of overload, are hard to quantify. An economic analysis of costs akin to the approach used by Gray and Shenoy [82] reveals the limitations of intuitive approaches. We reproduce below the analysis for prefetching done by Chandra et al. [37].

Let $p$ denote the threshold. Let $NWCost_{prefetch}$, $StorageCost$, $WaitCost$, $NWCost_{demand}$ denote the dollar costs of prefetching a byte over a wide-area link ($0.1 per MB), storing a byte on disk ($0.80 per GB per month), human wait time ($40 per hour), and the network cost of demand-fetching a byte over a wide-area link ($0.1 per MB) respectively. These numbers correspond to approximate valuations of the above in 2001. The values are related to each other through the threshold as

follows:

$$NWCost_{prefetch} + StorageCost \approx p \times (WaitCost + NWCost_{demand})$$

The left hand side represents the network and storage costs of prefetching a typical Web object while the second term in the product on the right hand side represents the cost of fetching the object on demand. It is useful to prefetch an object if the probability of accessing it is greater than the ratio of the prefetch and demand costs. If we assume that a typical Web document has size 10KB and that prefetching a file saves one second of human time over demand-fetching it, we obtain a value of approximately 0.01 for $p$. Note that this value is roughly two orders of magnitude smaller than typical thresholds used by prefetching systems. Moreover, falling costs of hardware resources will cause the threshold value to be further reduced by about 50% per year.

**Problem 2** : The value of the threshold varies with time. It varies over months and years with changing technology trends. It varies over hours with changes in network topology – for example, prefetching a file over a wireless network is more expensive than prefetching it over a local-area wired network, and diurnal load patterns. It also varies over seconds with fluctuations in network and server load over fine time-scales. The cost of interference is continuously changing and is a function of the demand load in the system.

**Problem 3** : Erring in choosing the value of the threshold can prove expensive. We demonstrated this risk earlier in Section 2.5 by means of an experiment involving prefetching on a Web server. Prefetching using static thresholds shifts the point where the system gets overloaded and effectively reduces available system capacity for demand load.

In summary, threshold-based approaches involve complex trade-offs across several factors that change over time. Thresholds in current prefetching systems are

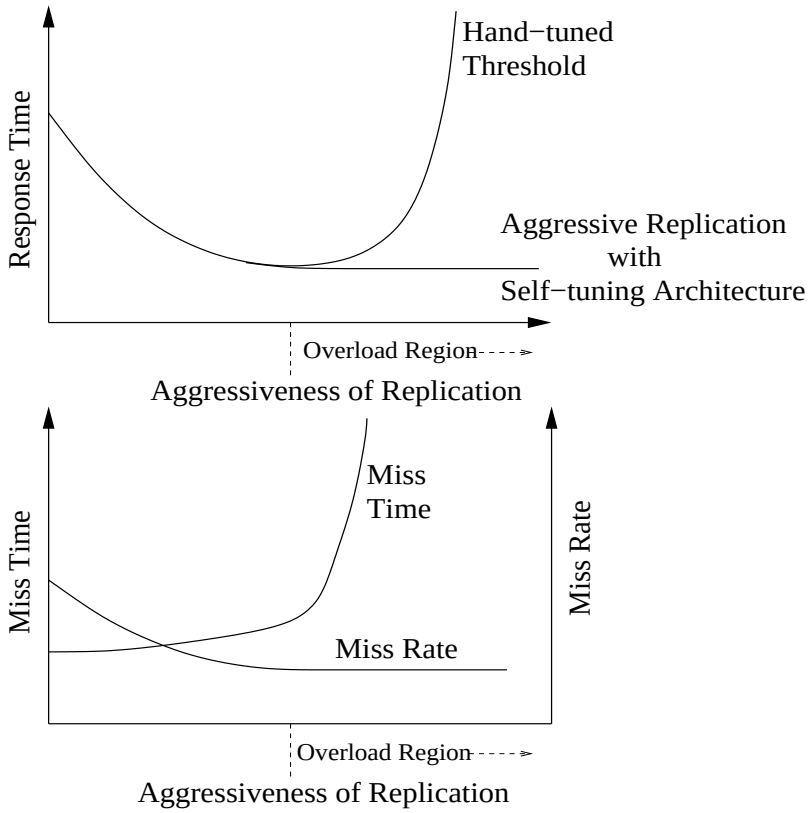



Figure 4.2: Utilization without risk

used only to limit bandwidth consumed by prefetch traffic. The key problem is to deal with system-wide interference at the server, network, and client in a self-tuning manner. Self-tuning support for speculative replication is conceptually captured by Figure 4.1. The diagram shows variation of response times (or unavailability or staleness) with the aggressiveness of speculative replication. An approach based on static thresholds gives benefits for conservative values on the x-axis. However, more aggressive values of the threshold result in overloading the system, leading to severe degradation of performance. The degradation due to static thresholds can be explained as follows. Speculative replication, at any level of aggressiveness, reduces

the miss rate observed by the application compared to one that does not support speculative replication. Typically, miss rates decreases rapidly initially by using up available spare capacity. Once the system is loaded to capacity, the reduction in miss rates taper off. Miss times, however, strictly increase throughout because of the extra load imposed by speculative replication. When the total load on the system is below capacity, the increase in miss times is gradual. The small increase in miss times combined with the rapid decrease in miss rate yields overall reductions in average response time in this region. When the aggressiveness of speculative replication drives the system beyond capacity, miss times start to sharply increase. The miss rate, however, does not decrease further in this region and may even start to increase due to inefficient utilization of system resources in regions of overload. The sharp increase in miss times in the overload region, due to interference between speculative and regular load, is what causes response times to drastically shoot up in this region. Thus, an incorrect choice of a static threshold to tune aggressiveness of speculative replication incurs a high cost that is a non-linear function of the "error". Consequently, application programmmers relying on manually tuned thresholds are forced to choose between being overly conservative and underutilizing available resources and being overly aggressive and incurring the risk of severe performance degradation.

In contrast, a self-tuning architecture for ASR relieves application programmers of the need to tune any thresholds. The system automatically reduces aggressiveness when demand load is high, and vice-versa, without manual intervention. Self-tuning support for ASR thus simplifies application design, gives increased benefits when there is spare capacity in the system, and maintains the system's robustness to the risk of overload.

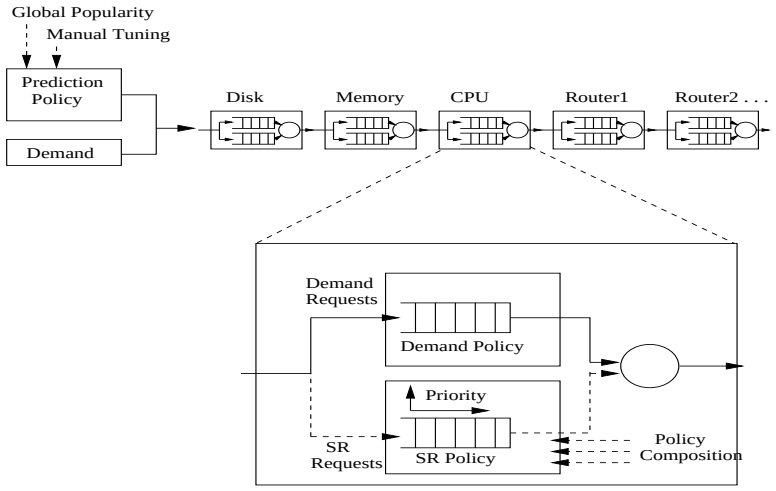



Figure 4.3: Separating Prediction from Scheduling

## 4.2 Separating Prediction from Scheduling

Mars achieves this self-tuning property by separating prediction in speculative replication systems from scheduling, i.e. separating what to replicate from how to allocate resources. Separation of concerns of policy and mechanism in computer systems is a well-aged wisdom that we continue to advocate in the context of ASR. Figure 4.2(b) illustrates this separation where a priority scheduler for resources separates demand requests from speculative replication. A replication policy predicts a current list of objects to push, and prioritizes more valuable objects. The list of prioritized objects that the predictor generates could be arbitrarily large and include objects with extremely low probabilities of access. The priority scheduler ensures that speculative requests are processed only if there are no outstanding demand requests to be processed, thereby ensuring non-interference as well as utilization of available resources.

For sake of comparision, Figure 4.2 illustrates the traditional threshold-based approach that is unable to prevent interference between speculative and regular load

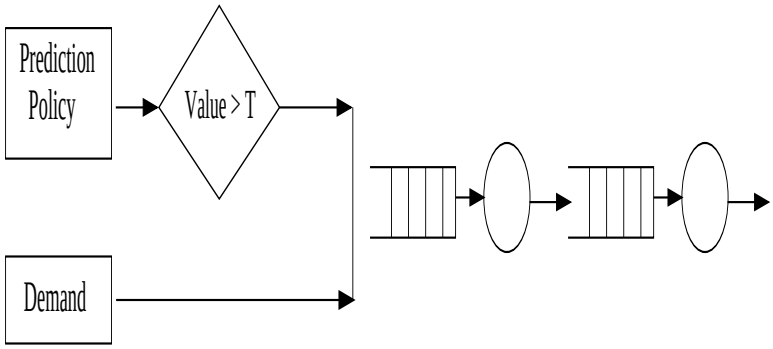



Figure 4.4: Traditional threshold-based replication system

because of the tight coupling of the prediction policy with the scheduling mechanism. The threshold, being a fixed value, is insufficient to adaptively prevent interference between speculative and regular load at each resource in the path of a request. As explained in the previous section, adjusting the threshold dynamically is complex, inefficient, and risk-prone. Instead, the Mars approach renders thresholds unnecessary by isolating speculative load from regular load at each resource in the system. Speculatively replicated objects are simply sorted in order of priority as decided by a policy appropriate for the system under consideration, and the scheduler at each resource automatically processes only as much speculative load as allowed by the spare capacity at that resource.

### 4.2.1 Realization of Mars

In order to achieve end-to-end isolation of speculative and demand traffic in Mars, the separation and prioritization must be done at every resource that could be a potential bottleneck, i.e. computing and storage at the server, replica and client, and network resources throughout the path from the server to the client.

In practical systems, such separation may be achieved in two ways described as follows. One could, as shown in Figure 4.2, augment each resource wth a low-

level scheduler that performs the separation. Several known solutions [38, 79, 124, 143, 161] exist for low-level CPU scheduling as well as disk scheduling that could be employed to achieve this separation at end-stations. For prioritization of network resources, solutions proposed in the literature such as Differentiated Services [21] could be employed at each router along the path. The advantage of using low-level per-resource schedulers is complete and clean isolation of speculative load from regular load. However, low-level solutions present significant deployability hurdles. It is hard to modify the internal of complex enterprise servers to accomodate schedulers for each resource. It is impractical to modify the large numbers of deployed routers to activate prioritized services.

Alternatively, one could realize Mars' conceptual architecture in a simpler and more practical manner at the cost of slightly imprecise separation of speculative and regular load. Such a realization can be achieved using end-to-end schedulers that abstract out components of the system at a coarser granularity and treat them as a black-box. For example, servers, networks, and clients can be treated as black-boxes to avoid modifying already deployed infrastructure and associated protocols. These system components can be monitored externally to guage the amount of regular load so that speculative load can be issued only when the system is determined to have sufficient spare capacity. An example of such an end-to-end scheduler is TCP Nice, introduced in the previous section, whose background abstraction can be used to prevent speculative traffic from interfering with regular traffic in the network without modifying individual routers. In the rest of this chapter, we focus on providing simple, deployable, and end-to-end solutions to realize Mars' architecture in practical systems. In particular, through the case study of NPS, a Web prefetching system, we show how a deployable realization of Mars is achieved without modifying servers, networks, clients, and existing protocols.

## 4.3 NPS : A Case Study

In order to validate the feasibility, benefits, and ease of deployability of Mars' architecture, we perform a case study of a Web prefetching system. Towards this end, we build *NPS*, a novel non-intrusive web prefetching system that (1) utilizes only spare resources to avoid interference between prefetch and demand requests at the server as well as in the network , and (2) is deployable without any modifications to servers, browsers, network or the HTTP protocol. NPS's self-tuning architecture eliminates the need for traditional "thresholds" or magic numbers typically used to limit interference caused by prefetching, thereby allowing applications to improve benefits and reduce the risk of aggressive prefetching.

A number of studies have demonstrated the benefits of web prefetching [50, 64, 86, 89, 111, 114, 140, 174]. And the attractiveness of prefetching appears likely to rise in the future as the falling prices of disk storage [55] and network bandwidth [139] make it increasingly attractive to trade increased consumption of these resources to improve response time and availability and thus reduce human wait time [37]. Despite these benefits, prefetching systems have not been widely deployed because of two concerns: interference and deployability. First, if a prefetching system is too aggressive, it may interfere with demand requests to the same service (self-interference) or to other services (cross-interference) and hurt overall system performance. Such interference may occur at the server, in the communication network or at the client. Second, if a system requires modifications to the existing HTTP protocol [72] , it may be impractical to deploy. The large number of deployed clients and networks in the Internet makes it difficult to change clients, and the increasing complexity of servers [85, 97, 99, 152, 189] [97] makes it difficult to change servers. What we therefore need is a prefetching system that (a) avoids interference at clients, networks, and servers and (b) does not require changes to the HTTP protocol and the existing infrastructure (client browsers, networks and

servers).

Through this case study, we make three contributions. First, we present NPS, a novel non-interfering prefetching system for the web that – (1) avoids interference by effectively utilizing only spare resources on the servers and the network and (2) is deployable with no modifications to the HTTP protocol and existing infrastructure. at the server, restricts the prefetch load imposed on it accordingly. To avoid interference in the underlying network, NPS uses TCP-Nice low-priority network transfers [173]. Finally, it uses a set of heuristics client. To work with existing infrastructure, NPS modifies HTML pages to include JavaScript$^{TM}$ code to issue prefetch requests, and wraps the server infrastructure with simple external modules that require no knowledge of, or no modifications to the internals of existing servers. Our measurements of the prototype under real web load trace indicate that NPS is both non-interfering and efficient under different network and server load conditions. For example, in our experiments on a heavily loaded network with little spare capacity, we observe that a threshold-based prefetching scheme causes response times to increase by a factor of 7 due to interference, whereas prefetching using NPS contains this increase to less than 30%.

Second, and on a broader note, we show that the self-tuning architecture of Mars eliminates the need for traditional "threshold" magic numbers that are typically used to limit the interference between speculative and regular requests. Moreover, the architecture is realizable for building practical replication systems through simple end-to-end mechanisms that require minimal changes to existing components of the system. In keeping with Mars' architecture, NPS divides prefetching into two separates tasks – (i) prediction and (ii) resource management. The predictor proposes prioritized lists of high-valued documents to prefetch. The resource manager limits the number of documents to prefetch and schedules the prefetch requests to avoid interference with demand requests and other applications. This separation

of concerns has three advantages – (i) it simplifies the design and deployment of prefetching systems by eliminating the need to choose appropriate thresholds for an environment and update them with changing conditions, (ii) it reduces the risk of interference caused by prefetching that relies on manually set thresholds, especially during periods of unanticipated high load, (iii) it increases the benefits of prefetching by prefetching more aggressively than would otherwise be safe during periods of low or moderate load. We believe that these advantages would also apply to speculative replication systems in many environments beyond the Web.

Third, we explore the design space for building a Web prefetching system, given the requirement of avoiding or minimizing changes to existing infrastructure. We find that it is straightforward to deploy prefetching that ignores the problem of interference, and it is not much more difficult to augment such a system to avoid server interference. Extending the system to also avoid network interference is more involved, but doing so appears feasible even under the constraint of not modifying current infrastructure. Unfortunately, we were unable to devise a method to completely eliminate prefetching's interference at existing clients: in our system prefetched data may displace more valuable data in a client cache. It appears that a complete solution may eventually require modifications at the client [34, 38, 143]. For now, we develop simple heuristics that reduce this interference.

The rest of this section is organized as follows. Section 4.3.1 discusses the requirements and architecture of a prefetching system. Sections 4.3.3, 4.3.4 and 4.3.5 present the building blocks for reducing interference at servers, networks and clients. Section 4.3.6 presents the prefetch mechanisms that we develop to realize the prefetching architecture. Section 4.3.7 discusses the details of our prototype and evaluation. Section 5.6 presents some related work and section 7.5 concludes.

### 4.3.1 Requirements and Alternatives

There appears to be a consensus among researchers on a high level architecture for prefetching in which a server sends a list of objects to a client and the client issues prefetch requests for the objects on the list [43, 129, 140] . This division of labor allows servers to use global object access patterns and service-specific knowledge to determine what should be prefetched, and it allows clients to filter requests through their caches to avoid repeatedly fetching objects. In this paper, we develop a framework for prefetching that follows this organization and that seeks to meet two other important requirements: self tuning resource management and deployability without modifying existing protocols, clients, proxies, or servers.

#### Resource Management

Services that prefetch should balance the benefits against the risk of interference. Interference can take the form of *self-interference*, where a prefetching service hurts its own performance by interfering with its demand requests, and *cross-interference*, where the service hurts the performance of other applications on the prefetching client, other clients, or both.

Limiting interference is essential because many prefetching services have potentially unlimited bandwidth demand, where incrementally more bandwidth consumption provides incrementally better service. For example, a prefetching system can improve hit rate and hence response times by fetching objects from a virtually unlimited collection of objects that have non-zero probabilities of access [28, 38], or by updating cached copies more frequently [46, 172, 174]. Interference can occur at any of the critical resources in the system.

- **Server:** Prefetching consumes extra resources on the server such as processing time, memory space and disk.

- **Network:** Prefetching causes extra data packets to be transmitted over the network, potentially increasing queuing delays and packet drops.
- **Client:** Prefetching results in extra processing at clients. Furthermore, aggressive prefetching can pollute a browser's memory and disk caches.

A common way of achieving balance between the benefits and costs of prefetching is to select a threshold and prefetch objects whose estimated probability of use before modification or eviction from the cache exceeds that threshold [64, 102, 140, 174]. There are at least two problems with such "magic number"-based approaches. First, it is difficult for even an expert to set thresholds to optimum values to balance costs and benefits—although thresholds relate closely to the benefits of prefetching, they have little obvious relationship to the costs of prefetching [37, 82]. Second, appropriate thresholds to balance costs and benefits may vary over time as client, network, and server load conditions change over seconds (e.g., changing workloads or network congestion [192]), hours (e.g., diurnal patterns), and months (e.g., technology trends [37, 139]).

Our goal is to construct a self-tuning resource module that prevents prefetch requests from interfering with demand requests. Such an architecture will simplify the design of prefetching systems by separating the tasks of prediction and resource management. Prediction algorithms may specify arbitrarily long lists of the most beneficial objects to prefetch sorted by benefit, and the resource management module issues requests for these objects and ensures that these requests do not interfere with demand requests or other system activities. In addition to simplifying system design, such an architecture could have two performance advantages over statically set prefetch thresholds. First, such a system can reduce interference – when resources are scarce, it would reduce prefetching aggressiveness. Second, such a system may increase the benefits of prefetching when resources are plentiful by allowing more aggressive prefetching than would otherwise be considered safe.

### Deployability

Many proposed prefetching mechanisms suggest modifying the HTTP/1.1 protocol [24, 59, 64, 140], to create a new request type for prefetching. An advantage of extending the protocol is that clients, proxies, and servers could then distinguish prefetch requests from demand requests and potentially schedule them separately to prevent prefetch requests from interfering with demand requests [59]. However, such mechanisms are not easily deployable because modifying the protocol implies modifying the widely-deployed infrastructure that supports the current protocol including existing clients, proxies, and servers. As web servers evolve and increase in their complexity, requests may traverse not only a highly optimized web server [141, 167, 178, 189] but also a number of other complex modules such as commercial databases, application servers or virtual machines for assembling dynamic content (e.g., Apache tomcat for executing Java Servlets and JavaServer pages), distributed cluster services [12, 85], and content delivery networks. Modifying servers to separate prefetch requests from demand requests maybe complex or infeasible under such circumstances.

If interference were not a concern, a simple prefetching system could easily be built with the present infrastructure, where clients can be made to prefetch without any modifications to the protocol. For example, servers can embed JavaScript code or a Java applet [73], to fetch specified objects over the network and load them into the browser cache. An alternative way is to add invisible frames to the demand content that include and thereby preload the prefetch content.

In this paper, we adapt such techniques to avoid interference while maintaining deployability.

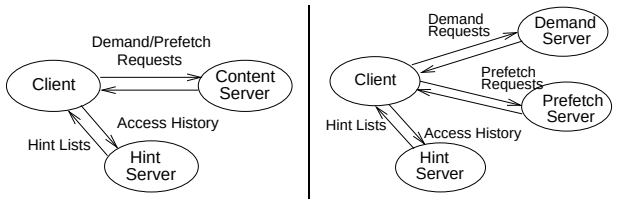
### 4.3.2 Architectural Alternatives

In this subsection, we present an overview of two alternative architectures to build a prefetching system. The high-level description in this section is intended only to provide a framework for discussing resource management strategies at the server, network, and client in sections 4.3.3 through 4.3.5. These architectures and resource management strategies are pertinent regardless of whether prefetching is implemented using a new protocol or by exploiting existing infrastructure. In Section 4.3.6, we describe how our implementation realizes one of these architectures in an easily deployable way.

We begin by making the following assumptions about client browsers:

- For easy deployability of the prefetching system, browsers should be unmodified.
- Browsers match requests to documents in their caches based on (among other parameters) the server name and the file name of the object on the server. Thus files of the same name served from different servers are considered to be different.
- Browsers may multiplex multiple client requests to a given server on one or more persistent connections [72].

Figure 4.5 illustrates what we call the one-connection and two-connection architectures respectively. In both architectures, clients send their access histories to the *hint server* and get a list of documents to prefetch. The hint server uses either online or offline prediction algorithms to compute the hint lists consisting of the most probable documents that the users might request in the future.




(a) One Connection (b) Two Connection

Figure 4.5: Design Alternatives for a Prefetching System

### One Connection

In the one connection architecture (Figure 4.5(a)), a client fetches both demand and prefetch requests from the same content server. Since browsers multiplex requests over established connections to servers, and since browsers do not differentiate between demand and prefetch requests, each TCP connection may interleave prefetch and demand requests and responses.

Sharing connections can cause prefetch requests to interfere with demand requests for network and server resources. If interference can be avoided, this system is easily deployable. In particular, objects fetched from the same server share the domain name of the server. So, unmodified client browsers can use cached prefetched objects to service demand requests.

### Two Connection

In the two connection architecture(Figure 4.5(b)), a client fetches demand and prefetch requests from different servers or from different ports on the same server. This architecture thus segregates demand and prefetch requests on separate network connections.

Although the two connection architecture simplifies the mechanisms for reducing interference at the server by segregation, this solution appears to complicate the deployability of the system. Objects with the same names fetched from different servers are considered different by the browsers. So, browsers can not directly use

the prefetched objects to service demand requests.

#### Comparison

In the following sections, we show how to address the limitations of both architectures.

- Some of the techniques we develop for avoiding interference are useful for the one connection architecture, but some are less so. In particular, our strategy for reducing interference at servers is based on end-to-end performance and is equally applicable to the one and two connection architectures. Conversely, the techniques we use to avoid network interference appear much easier to apply to the two-connection than the one-connection architecture.
- Despite the apparent deployability challenges to the two connection architecture discussed above, we find that the same basic technique we use to make unmodified browsers prefetch data for the one connection architecture can be adapted to support the two connection architecture as well.

We conclude that both architectures are tenable in some circumstances. If server load is the primary concern and if network load is known not to be a major issue, then the one connection prototype may be simpler than the two connection prototype. At the same time, the two connection prototype is feasible and deployable and manages both network and server interference. Given that networks are a globally shared resource, we recommend the use of two connection architecture in most circumstances.

### 4.3.3 Server Interference

An ideal system for avoiding server interference would cause no delay to demand requests in the system and utilize significant amounts of any spare resources on servers for prefetching. Such a system needs to cope with, and take advantage of,

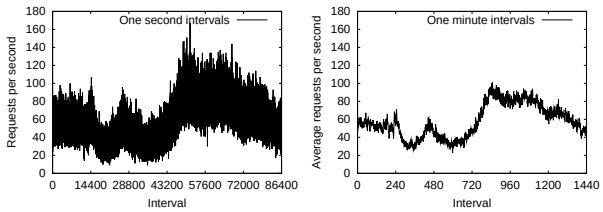



Figure 4.6: Server loads averaged over (a) 1-second and (b) 1-minute intervals for the IBM sporting event workload.

changing workload patterns over various time scales. HTTP request traffic arriving at a server often is bursty with the burstiness being observable at several scales of observation [51] and with peak rates exceeding the average rate by factors of 8 to 10 [130]. For example, Figure 4.6 shows the request load on an IBM server hosting a major sporting event during 1998 averaged over 1-second and 1-minute intervals. It is crucial for the prefetching system to be responsive to such bursts to balance utilization and risk of interference. There are a variety of ways to prevent prefetch requests from interfering with demand requests at servers.

**Local scheduling** Server scheduling can help use the spare capacity of existing infrastructure for prefetching in a non-interfering manner. In principle, existing schedulers for processor, memory [102, 107, 143], and disk [125] could prevent low-priority prefetch requests from interfering with high-priority demand requests. Furthermore, as these schedulers are intimately tied to the operating system, they should be highly efficient in delivering whatever spare capacity exists to prefetch requests even over fine time scales. Note that local scheduling is equally applicable to both one- and two-connection architectures.

For many services, however, server scheduling may not be easily deployable for two reasons. First, although several modern operating systems support process

schedulers that can provide strict priority scheduling, few provide memory, cache or disk schedulers that isolate prefetch requests from demand requests. Second, even if an operating system provides the needed support, existing servers would have to be modified to differentiate between prefetch and demand requests with scheduling priorities as they are serviced [14]. This second requirement appears particularly challenging given the increasing complexity of servers, in which requests may traverse not only a highly-tuned web server [141, 167, 178, 189] but also a number of other complex modules such as commercial databases, application servers or virtual machines for assembling dynamic content (e.g., Apache tomcat for executing Java Servlets and JavaServer pages), distributed cluster services [12, 85], and content delivery networks.

**Separate prefetch infrastructure** An intuitively simple way of avoiding server interference is to use separate servers to achieve complete isolation of prefetch and demand requests. In addition to the obvious strategy of providing separate demand and prefetch machines in a centralized cluster, a natural use of this strategy might be for a third-party "prefetch distribution network" to supply geographically distributed prefetch servers in a manner analogous to existing content distribution networks. Note that this alternative is not available to the one-connection architecture.

However, separate infrastructure needs extra hardware and hence may not be an economically viable solution for many web sites.

**End-to-end monitoring** A technique based on end-to-end monitoring estimates the overall load (or spare capacity) on the server by periodically probing the server with representative requests and measuring the response times of the replies. Low response times indicate that the server has spare capacity and high response times indicate that the server is loaded. Based on such an estimate, the monitor utilizes

the spare capacity on the server by controlling the number and aggressiveness of prefetching clients.

An advantage of end-to-end monitoring is that it requires no modifications to existing servers. Furthermore, it can be used by both one- and two- connection prefetching architectures. The disadvantage of such an approach is that its scheduling precision is likely to be less than that of a local scheduler that has access to the internal state of the server and operating system. Moreover, an end-to-end monitor may not be responsive enough to bursts in load over fine time scales.

In the following subsubsections, we discuss issues involved in designing an end-to-end monitor in greater detail, present our simple monitor design, and evaluate its efficacy in comparison to server scheduling.

### End-to-end Monitor Design

Figure 4.7 illustrates the architecture of our monitor-controlled prefetching system. The monitor estimates the server's spare capacity and sets a *budget* of prefetch requests permitted for an interval. The hint server adjusts the load imposed by prefetching on the server by ensuring that the sum across the hint lists returned to clients does not exceed the budget. Our monitor design must adress two issues: (i) budget estimation and (ii) budget distribution across clients.

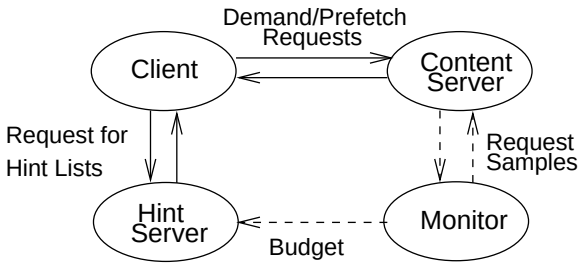



Figure 4.7: A Monitored Prefetching System

**Budget estimation** The monitor periodically probes the server with HTTP requests to representative objects and measures the response times. The monitor increases the budget when the response times are below the objects' *threshold* values and decreases the budget otherwise.

As probing is an intrusive technique, choosing an appropriate rate of probing is a challenge. A high rate makes the monitor more reactive to load on the server, but also adds extra load on the server. On the other hand, a low rate makes the monitor react slowly, and can potentially lead to interference to the demand requests. Similarly, the exact policy for increasing and decreasing the budget must balance the risk of causing interference against underutilization of spare capacity.

**Budget distribution** The goal of this task is to distribute the budget among the clients such that (i) the load due to prefetching on the server is contained within the budget for that epoch and is distributed uniformly over the interval, (ii) a significant fraction of the budget is utilized over the interval, and (iii) clients are responsive to changing load patterns at the server. The two knobs that the hint server can manipulate to achieve these goals are (i) the size of the hint list returned to the clients and (ii) the subset of clients that are given permission to prefetch. This flexibility provides a freedom to choose from many policies.

### Monitor Prototype

Our prototype uses simple, minimally tuned policies for budget estimation and budget distribution. Future work may improve the performance of our monitor.

The monitor probes the server in epochs, each approximately 100 ms long. In each epoch, the monitor collects a response time sample for a representative request. In the interest of being conservative – choosing non-interference even at the potential cost of reduced utilization – we use an additive increase(increase by 1), multiplicative decrease (reduce by half) policy. AIMD is commonly used in network

congestion control [103] to conservatively estimate spare capacity in the network and be responsive to congestion. If in five consecutive epochs, the five response time samples lie below a threshold, the monitor increases the budget by 1. While taking the five samples, if any sample exceeds the threshold, the monitor sends another probe immediately to check if the sample was an outlier. If even the new sample exceeds the threshold, indicating a loaded server, the monitor decreases the budget by half and restarts collecting the next five samples.

In our simple prototype, we manually supply the representative objects's threshold response times. However, it is straightforward because of the predictable pattern in which response times vary with load on server systems – a nearly constant value of response time for low load followed by a sharp rise beyond the "knee" for high load. As part of our future work, we intend to make the monitor automatically pick thresholds in a self-tuning manner.

The hint server distributes the current budget among client requests that arrive in that epoch. We choose to set the hint list size to the size of one document (a document corresponds to a HTML page and all embedded objects). Our policy lets clients to return quickly for more hints and thus be more responsive to changing load patterns on the server. Note that returning larger hint lists would reduce the load on the hint server, but it would reduce the system's responsiveness and its ability to avoid interference. We control the number of simultaneously prefetching clients, and thus the load on the server, by returning to some clients a hint list of zero size and a directive to wait until the next epoch to fetch the next hint list. For example, if $B$ denotes the budget in the current epoch, and $N$ the expected number of clients in that epoch, $D$ the number of files in a document, and $\tau$ the epoch length, the hint server accepts a fraction $p = min(1, \frac{B \cdot \tau}{N \cdot D})$ of requests to prefetch on part of clients in that epoch and returns hintlists of zero length for other requests. Note that other designs are possible. For example, the monitor can integrate with the

prefetch prediction algorithm to favor prefetching by clients for which the predictor can identify high-probability items and defer prefetching by clients for which the predictor identifies few high-value targets.

Since the hint server does not a priori know the number of client requests that will come in an epoch, it estimates that value with the number of requests that come in the previous epoch. If more than the estimated number of requests arrive in a epoch, the hint server replies with list of size zero and a directive to retry in the next epoch to those extra requests. If fewer clients arrive, some of the budget can get wasted. However, in the interest of avoiding interference, we choose to allow such wastage of budget.

In the following Section 4.3.3, we evaluate the performance of our prototype with respect to the goals of reducing interference and reaping significant spare bandwidth and compare it with the other resource management alternatives.

### Server Interference Experiments

In evaluating resource management algorithms, we are mainly concerned with interference that prefetching could cause and less with the benefits obtained. We therefore abstract away prediction policies used by services by prefetching sets of dummy data from arbitrary URLs at the server. The goal of the experiments is to compare the effectiveness of different resource management alternatives in avoiding server interference against the ideal case (when no prefetching is done) with respect to the following metrics: (i) *cost:* the amount of interference in terms of demand response times and (ii) *benefit:* the prefetch bandwidth.

We consider the following resource management algorithms for this set of experiments:

1. No-Prefetching: Ideal case, when no prefetching is done or when we use a separate prefetching infrastructure.

2. No-Avoidance: Prefetching with no interference avoidance with fixed aggressiveness. We set the aggressiveness by setting *pfrate*, which is the number of documents prefetched for each demand document. For a given service, a given prefetch threshold will correspond to some average *pfrate*. We use fixed *pfrate* values of 1 and 5.
3. Scheduler: As a simple local server scheduling policy, we choose *nice*, the process scheduling utility in Unix. We again use fixed *pfrate* values of 1 and 5. This simple server scheduling algorithm is only intended as a comparison; more sophisticated local schedulers may better approximate the ideal case.
4. Monitor: We perform experiments for two threshold values of 3ms and 10ms.

For evaluating algorithms 2 and 4, we set up one server serving both demand and prefetch requests. These algorithms are applicable in both one connection and two connection architectures. Our prototype implementation of algorithm 3 requires that the demand and prefetch requests be serviced by different processes and thus is applicable only to the two connection architecture. We use two different servers listening on two ports on the same machine, with one server run at a lower priority using the Linux *nice*. Note that the general local scheduling approach is equally applicable to the one-connection architecture with more intrusive server modifications.

Our experimental setup includes Apache HTTP server [11] running on a 450MHz Pentium II, with 128MB of memory. To generate the client load, we use httperf [132] running on four different Pentium III 930MHz machines. All machines run the Linux operating system.

We use two workloads in our experiments. Our first workload generates demand requests to the server at a constant rate. The second workload is a one hour subset of the IBM sporting event server trace, whose characteristics are shown in Figure 4.6. We scale up the trace in time by a factor of two, so that requests are

generated at twice the original rate, as the original trace barely loads our server.

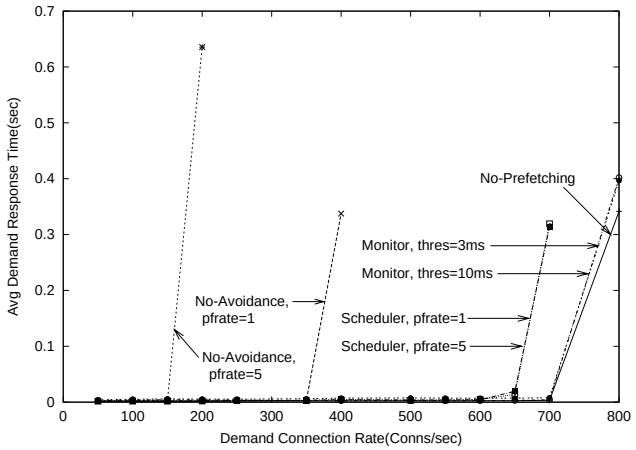



Figure 4.8: Effect of prefetching on demand throughput and response times with various resource management policies

**Constant workload** Figure 4.8 shows the demand response times with varying demand request arrival rate. The graph shows that both Monitor and Scheduler algorithms closely approximate the behavior of No-Prefetching in not affecting the demand response times. Whereas, the No-Avoidance algorithm with fixed *pfrate* values significantly damages both the demand response times and the maximum demand throughput.

Figure 4.9 shows the bandwidth achieved by the prefetch requests and their effect on the demand bandwidth. The figure shows that No-Avoidance adversely affects the demand bandwidth. Conversely, both Scheduler and Monitor reap spare bandwidth for prefetching without much decrease in the demand bandwidth. Further, at low demand loads, a fixed pfrate prevents No-Avoidance from utilizing the full available spare bandwidth. The problem of too little prefetching when demand load is low and too much prefetching when demand load is high illustrates the problem with existing threshold strategies. As hoped, the Monitor tunes prefetch aggressiveness of the clients such that essentially all of the spare bandwidth is uti-

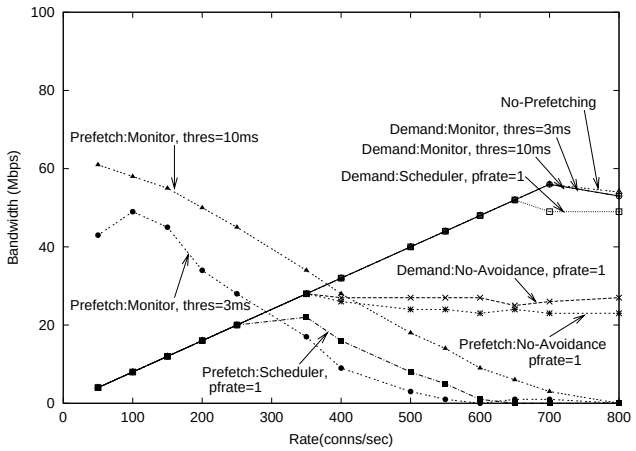



Figure 4.9: Prefetch and demand bandwidths achieved by various algorithms

lized.

**IBM server trace** In this set of experiments, we compare the performance of the four algorithms for the IBM server trace. Figure 4.10 shows the demand response times and prefetch bandwidth in each case. The graph shows that the No-Avoidance case affects the demand response times significantly as *pfrate* increases. The Scheduler and Monitor cases have less adverse effects on the demand response times.

These experiments show that resource management is an important component of a prefetching system because overly aggressive prefetching can significantly hurt demand response time and throughput while timid prefetching gives up significant bandwidth. They also illustrate a key problem with constant non-adaptive magic numbers in prefetching such as the threshold approach that is commonly proposed. The experiments also provide evidence of the effectiveness of the monitor in tuning prefetch aggressiveness of clients to reap significant spare bandwidth while keeping interference at a minimum.

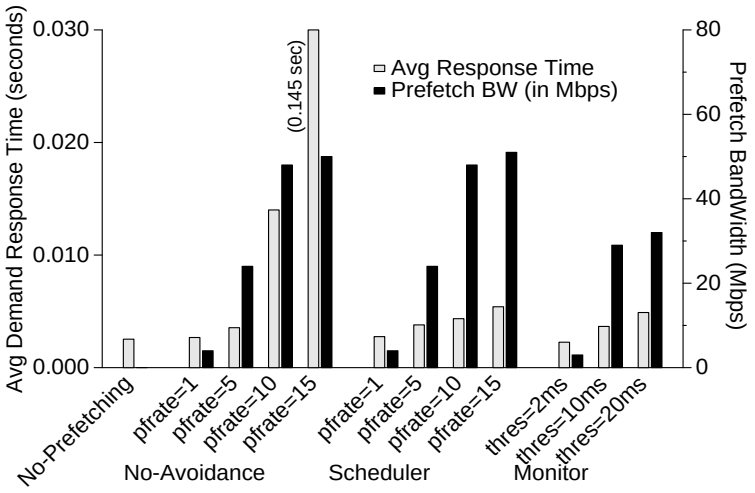



Figure 4.10: Performance of No-Avoidance, Scheduler and Monitor schemes on the IBM server trace

### 4.3.4 Network Interference

Mechanisms to reduce network interference could, in principle, be deployed at clients, intermediate routers, or servers. For example, clients can reduce the rate at which they receive data from the servers using TCP flow control mechanisms [163]. However, it is not clear how to set the parameters to such mechanisms or how to deploy them given existing infrastructure. Prioritization in routers that provide differentiated service to prefetch and demand packets can avoid interference effectively [21]. However, router prioritization is not easily deployable in the near future. We focus on server based control because of the relative ease of deployability of server based mechanisms and their effectiveness in avoiding both self- and cross-interference.

In particular, we use TCP-Nice at the server. TCP-Nice, introduced in the previous chapter, uses a congestion control mechanism at the sender that is specifically designed to support background transfers. Background connections using Nice operate by utilizing only spare bandwidth in the network. They react more sensi-

tively to congestion and backoff when a possibility of congestion is detected, giving way to foreground connections. In the previous chapter, we showed a small upper bound on the network interference caused by Nice under a simple network model. Furthermore, the experimental evidence under wide range of conditions and workloads showed that Nice causes little or no interference and at the same time reaps a large fraction of the spare capacity in the network.

Nice is deployable in the two connection context without modifying the internals of servers by configuring systems to use Nice for all connections made to the prefetch server. A prototype of Nice runs on Linux currently. We also built a user-level version of Nice, that can be easily installed without requiring any kernel modifications and has comparable performance to the kernel version. Though both the kernel and user-level versions currently run on Linux, we believe it should be straightforward to port them to other operating systems. An alternative to writing and maintaining different versions of Nice for different operating systems is to place a Linux machine running Nice in front of the prefetch server and make the Linux machine serve as a reverse proxy or a gateway.

It appears to be more challenging to use Nice in the one connection case. In principle, the Nice implementation allows flipping a connection's congestion control algorithm between standard TCP (when serving demand requests) and Nice (when serving prefetch requests). However, using this approach for prefetching faces a number of challenges: (1) Flipping modes causes packets already queued in the TCP socket buffer to inherit the new mode. Thus, demand packets queued in the socket buffer may be sent at low-priority while prefetch packets may be sent at normal-priority, thus causing network interference. Ensuring that demand and prefetch packets are sent in the appropriate modes would require an extension to Nice and a fine-grained coordination between the application and the congestion control implementation. (2) Nice is designed for long network flows. It is not clear if flipping

back and forth between congestion control algorithms will still avoid interference and gain significant spare bandwidth. (3) HTTP/1.1 pipelining requires replies to be sent in the order requests were received, so demand requests may be queued behind prefetch requests, causing demand requests to perceive increased latencies. One way to avoid such interference may be to quash all the prefetch requests queued in front of the demand request. For example, we could send a small error message (eg. HTTP response code 307 – "Temporary Redirect" with a redirection to the original URL) as a response to the quashed prefetch requests.

Based on these challenges, it appears simpler to use the two connection architecture when the network is a potential bottleneck. A topic for future work is to explore these challenges and determine if a deployable one connection architecture that avoids network interference can be devised.

### 4.3.5 Client Interference

Prefetching may interfere with the performance of a client in at least two ways. First, prefetch requests consume processing cycles and may, for instance, delay rendering of demand pages. Second, prefetched data may displace demand data from the cache and thus hurt demand hit rates for the prefetching service or other services.

As with the interference at the server discussed above, interference between client processes could, in principle, be addressed by modifying the client browser (and, perhaps, the client operating system) to use a local processor scheduler to ensure that processing of prefetch requests never interferes with processing of demand requests. Lacking that option, we resort to a simpler approach: as described in Section 4.3.6, we structure our prefetch mechanism to ensure that processing prefetch requests does not begin until after the loading and rendering of the demand page, including all inline images and recursive frames. Although this approach will not help reduce cross-interference with other applications at the client, it may

avoid a potentially common case of self-interference of the prefetches triggered by a page delaying the rendering of that page. Similarly, a number of storage scheduling algorithms exist that balance caching prefetched data against caching demand data [34, 38, 107, 143]. Unfortunately, all of these algorithms require modifications to the cache replacement algorithm.

Because we assume that the client cannot be modified, we resort to two heuristics to limit cache pollution caused by prefetching. First, in our system, services place a limit on the ratio of prefetched bytes to demand bytes sent to a client. Second, services can set the `Expires` HTTP header to a value in the relatively near future (e.g., one day in the future) to encourage clients to evict prefetched document earlier than they may otherwise have done. These heuristics have an obvious disadvantage: they resort to magic numbers similar to those in current use, and they suffer from the same potential problems: if the magic numbers are too aggressive, prefetching services will interfere with other services, and if they are too timid, prefetching services will not gain the benefits they might otherwise gain. Fortunately, there is reason to hope that performance will not be too sensitive to this parameter. First, disks are large and growing larger at about 100% per year [55] and relatively modest-sized disks are effectively infinite for many client web cache workloads [174]. So, disk caches may absorb relatively large amounts of prefetch data with little interference. Second, hit rates fall relatively slowly as disk capacities shrink [28, 174], which would suggest that relatively large amounts of polluting prefetch data will have relatively small effects on demand hit rate.

Figure 4.11 illustrates the extent to which our heuristics can limit the interference of prefetching on hit rates. We use the 28-day UCB trace of 8000 unique clients from 1996 [84] and simulate the hit rates of 1 MB, 10 MB and 30 MB per-client caches. Note that these cache sizes are small given, for example, Internet Explorer's defaults of using 3% of a disk's capacity (e.g., 300 MB of a 10 GB disk)

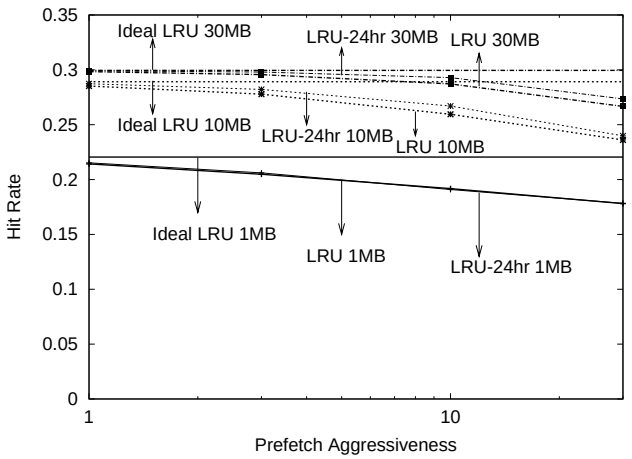



Figure 4.11: Effect of prefetching on demand hit rate

for web caching. On the x-axis, we vary the number of bytes of dummy prefetch data per byte of demand data that are fetched after each demand request. In this experiment, 20% of services use prefetching at the specified aggressiveness and the remainder do not, and we plot the demand hit rate of the *non*-prefetching services. Ideally, these hit rates should be unaffected by prefetching. As the graph shows, hit rates fall gradually as prefetching increases, and the effect shrinks as cache sizes get larger. For example, if a client cache is 30 MB and 20% of services prefetch aggressively enough that each prefetches ten times as much prefetch data as the client references demand data, demand hit rates fall from 29.9% to 28.7%.

### 4.3.6 Prefetching Mechanism

Figures 4.12 and 4.13 illustrate the key components of the one and two connection architectures. The one-connection mechanism consists of an unmodified client, a content server that serves both demand and prefetch requests, a munger that modifies content on the content server to activate prefetching and a hint server that gives out hint lists to the client to prefetch. The hint server also includes a monitor that probes the content server and estimates the spare capacity at the server and

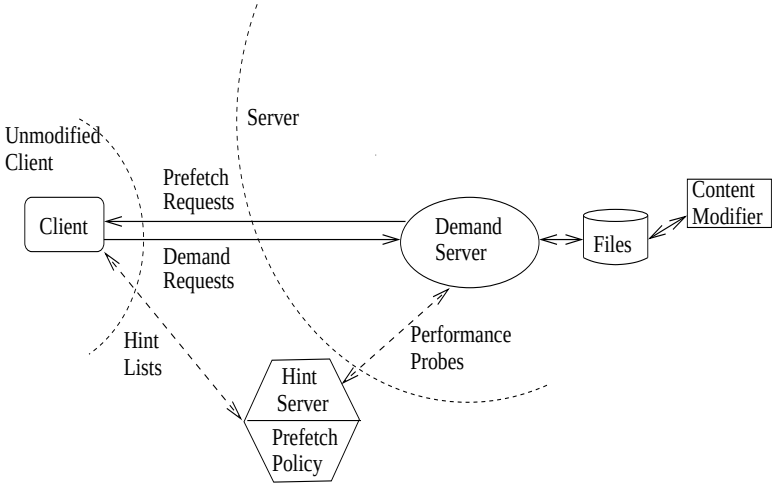



Figure 4.12: Prefetching mechanism for the one connection architecture

accordingly controls the number of prefetching clients.

The two-connection prototype, along with the components above, also consists of a prefetch server that is a copy of the demand server (running either on a separate machine or on a different port on the same machine) and a front-end that intercepts certain requests to the demand server and returns appropriate redirection objects as described later, thereby obviating any need to modify the original demand server.

In the following subsubsections, we describe the prefetching mechanisms for the one and two connection architectures.

#### One-connection

**Content modification** The munger augments each HTML document with pieces of JavaScript and HTML code that cause the client to prefetch.

#### On demand fetch

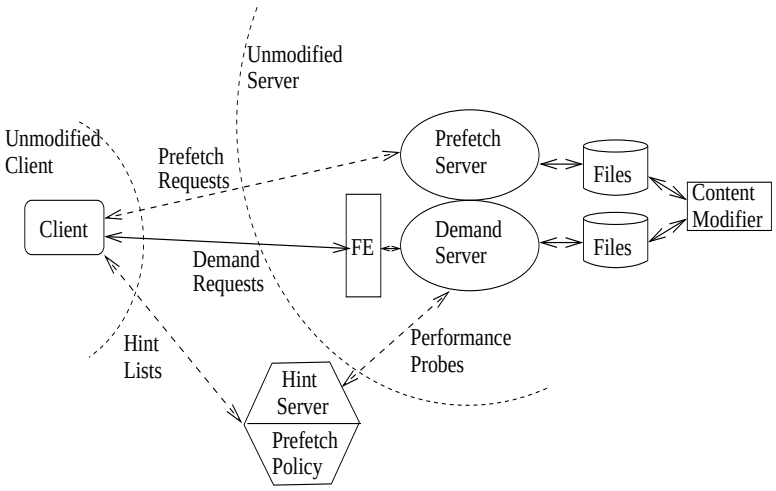



Figure 4.13: Prefetching mechanism for the two connection architecture

1. Client requests an augmented HTML document.
2. When an augmented HTML document (Figure 4.14) finishes loading into the browser, the `pageOnLoad()` function is called. This function calls `getPfList()`, a function defined in `pfalways.html` (Figure 4.15). The file `pfalways.html` is loaded within every augmented HTML document. `pfalways.html` is cacheable and hence does not need to be fetched everytime a document gets loaded.
3. `getPfList()` sends a request for `pflist.html` to the hint server with the name of the enclosing document, the name of the previous document in history (the enclosing document's referer) and TURN=1 as extra information embedded in the URL.
4. The hint server receives the request for `pflist.html`. Since the client fetches a `pflist.html` for each HTML document (even if the HTML document is found in the cache), the client provides the hint server with a history of accesses to aid in predicting hint lists. In Figure 4.15, PFCOOKIE contains the present access (`document.referrer`) and the last access (`prevref`) by the client. The

hint updates the history and predicts a list of documents to be prefetched by the client based on that client's history and the global access patterns. It puts these predictions into the response `pflist.html` such as shown in 4.16, which it returns to the client.

5. `pflist.html` replaces `pfalways.html` on the client. After `pflist.html` loads, the `preload()` function in its body preloads the documents to be prefetched from the prefetch server (which is same as the demand server in the one connection case).
6. After all the prefetch documents are preloaded, the `myOnLoad()` function calls `getMore()` that replaces the current `pflist.html` by fetching a new version with `TURN=TURN`+1.

Steps 5 and 6 repeat until the hint server has sent everything it wants, at which point the hint server returns a `pflist.html` with no `getMore()` call. When there is not enough budget left at the server, the hint server sends a `pflist.html` with no files to prefetch and a delay, after which the `getMore()` function gets called. The information TURN breaks the (possibly) long list of prefetch suggestions into a "chain" of short lists.

### On demand fetch of a prefetched document

1. The client browser simply fetches it from the cache as if it is a cache hit.

### Two-connection

The two-connection prototype employs the same basic mechanism for prefetching as the one-connection prototype. However, since browsers identify cached documents using both the server name and document name, documents fetched from prefetch server are not directly usable to serve demand requests. In order to fix this problem, we modify step 6 such that before calling `getMore()`,

```
<HTML> <HEAD> <! -- existing header goes here -- >
<SCRIPT LANGUAGE="JavaScript">
function pageOnLoad() {
 myiframe.getPFlist(document.referrer);
} </SCRIPT> </HEAD> <BODY>

<! -- existing body goes here -- >
if(null == window.onload) {
 window.onload = pageOnLoad();}
else {
 var origfn = window.onload;
 window.onload = function(){origfn();pageOnLoad();};}

<IFRAME SRC="pfalways.html" name="myiframe"
        width=0 height=0 frameborder=0>
</IFRAME> </BODY> </HTML>
```

Figure 4.14: Augmentation of HTML pages

```
<HTML> <HEAD> <SCRIPT LANGUAGE="JavaScript">
function getPFList(var prevref) {
 document.location="HINT-SERVER/pflist.html+PCOOKIE="
        + document.referrer + "+" + prevref + TURN=1;
 document.close();
} </SCRIPT> </HEAD> </HTML>
```

Figure 4.15: pfalways.html

6.a The `myOnLoad()` function (Figure 4.16) requests a *wrapper* (redirection object) from the demand server for the document that was prefetched.

6.b The frontend intercepts the request (based on the referer field) and responds with the wrapper (Figure 4.17) that loads the prefetched document in response to a client's demand request.

The prefetch server serves a modified copy of the content on the demand server. Note that the relative links in a webpage on the demand server point to pages on demand server. Hence, all relative links in the prefetch server's content are changed to absolute links, such that when client clicks on a link in the prefethed

```
<HTML> <HEAD> <SCRIPT LANGUAGE="JavaScript">

function myOnLoad() { //exeutes after body loads
 preload("DEMAND-SERVER/c.html"); //For two-conn only
 getMore() ;
}
function getMore() {
 document.location="HINT-SERVER/pflist.html +
                   PCOOKIE=" + document.referrer +
                   "+" + prevref + "+" + "TURN=2";
 document.close();
}

var myfiles=new Array()
function preload(){
 for (i=0;i<preload.arguments.length;i++){
  myfiles[i]=new Image() ;
  myfiles[i].src=preload.arguments[i] ;
 }
} </SCRIPT> </HEAD>

<BODY onload="myOnLoad()">
<SCRIPT LANGUAGE="JavaScript">
preload("PREFETCH-SERVER/a.jpg",
        "PREFETCH-SERVER/b.jpg",
        "PREFETCH-SERVER/c.html");
</SCRIPT> </BODY> </HTML>
```

Figure 4.16: An example pflist.html returned by the hint server

web page, the request is sent to the demand server. Also, all absolute links to inline objects in the page are changed to be absolute links to the prefetch server, so that prefetched inline objects are used. Since prefetch and demand servers are considered as different domains by the client browser, JavaScript security models [137] prevent scripts in prefetched documents to access private information of the demand documents and vice versa. However, to fix this problem, JavaScript allows us to explicitly set the `document.domain` property of each HTML document to a common suffix of prefetch and demand servers. For example, for servers `demand.cs.utexas.edu` and `prefetch.cs.utexas.edu`, all the HTML documents can set their `document.domain` property to `cs.utexas.edu`.

```
<HTML> <SCRIPT LANGUAGE="JavaScript">
if (document.referrer.indexOf ("pflist") < 0)
          document.location="PREFETCH-SERVER/c.html";
document.close();
</SCRIPT> </HTML>
```

Figure 4.17: Wrapper for c.html, stored in cache as DEMAND-SERVER/c.html

On demand fetch of a prefetched document: (i) a hit results for the wrapper in the cache, (ii) at the loading time, the wrapper replaces itself with the prefetched document from the cache, (iii) inline objects in the prefetched document point to objects from the prefetch server and hence are found in the cache as well, and (iv) links in the prefetched document point to the demand server.

This mechanism has two limitations. First, prefetched objects might get evicted from the cache before their wrappers. In such a case, when the wrapper loads for a demand request, a new request will be sent to the prefetch server. Since sending a request to the prefetch server in response to a demand request could cause undesirable delay, we reduce such occurrences by setting the expiration time of the wrapper to a value smaller than the expiration of the prefetched object itself. Second, but not a significant limitation is that some objects may be fetched twice, once as demand and once as prefetch objects as the browser cache considers them as different objects.

### Prediction Algorithm

For our experiments, we use prediction by partial matching [47] (PPM-$n/w$) to generate hint lists for prefetching. The algorithm uses a client's $n$ most recent requests to the server for non-image data to predict URLs that will appear during a subsequent window that ends after the $w$'th non-image request to the server. Our prototype uses $n$=2 and $w$=10.

In general, the hint server can be made to use any prediction algorithm. It can be made to use standard algorithms proposed in the literature [64, 70, 86, 140] or others that utilize more service specific information such as a news site that prefetches stories relating to topics that interest a given user.

#### Alternatives

We explored other alternatives for prefetching in the two-connection architecture. We could have used a Java Applet instead of the JavaScript in Figure 4.14. One could also use a zero-pixel frame that loads the prefetched objects instead of JavaScript. The refresh header in HTTP/1.1 could be exploited to iteratively prefetch a list of objects by setting the refresh time to a small value.

As an alternative to using wrappers, we also considered maintaining state explicitly at the client to store information about whether a document has already been prefetched. Content could be augmented with a script to execute on a hyperlink's `onClick` event that checks this state information before requesting a document from the demand server or prefetch server. Similar augmentation could be done for inline objects. Tricks to maintain state on the client can be found in [149].

### 4.3.7 Prototype and Evaluation

Our prototype uses the two connection architecture whose prefetching mechanism is shown in Figure 4.13. We use Apache 2.0.39 as the server, hosted on a 450MHz Pentium II, serving demand requests on one port and prefetch requests on the other. As an optimization, we implemented the frontend as a module within the Apache server rather than as a separate process. The hint server is implemented in Java and runs on a separate machine with 932 MHz Pentium III proessor, and connects to the server over a 100 Mbps LAN. The hint server uses prediction lists generated offline using the PPM algorithm [140] over a complete 24 hour IBM server trace.

The monitor runs as a separate thread of the hint server on same machine. The content munger is also written in Java and modifies the content offline (as shown in Figure 4.14). We have successfully tested our prefetching system with popular web browsers inluding Netscape, Internet Explorer, and Mozilla. [1]

### 4.3.8 End to End Performance

In this section, we evaluate NPS under various setups and evaluate the importance of each component in our system. In all setups, we consider three cases: (1) No-Prefetching, (2) No-Avoidance scheme with fixed *pfrate*, and (3) NPS (with Monitor and TCP-Nice). In these experiments, the client connects to the server over a wide area network through a commercial cable modem link. On an unloaded network, the round trip time from the client to the server is about 10 ms and the bandwidth is about 1 Mbps.

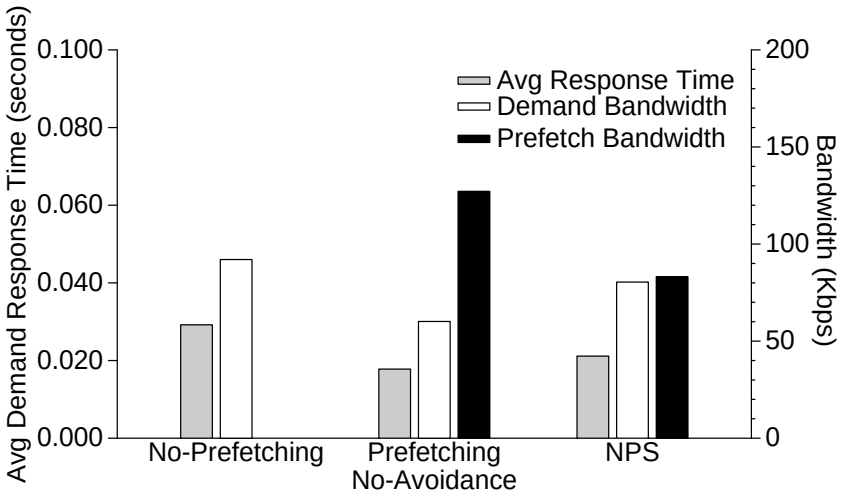



Figure 4.18: Effect of prefetching on demand response times with unloaded resources

[1] Source code for NPS prototype can be downloaded from *http://www.cs.utexas.edu/users/rkoku/RESEARCH/NPS/*

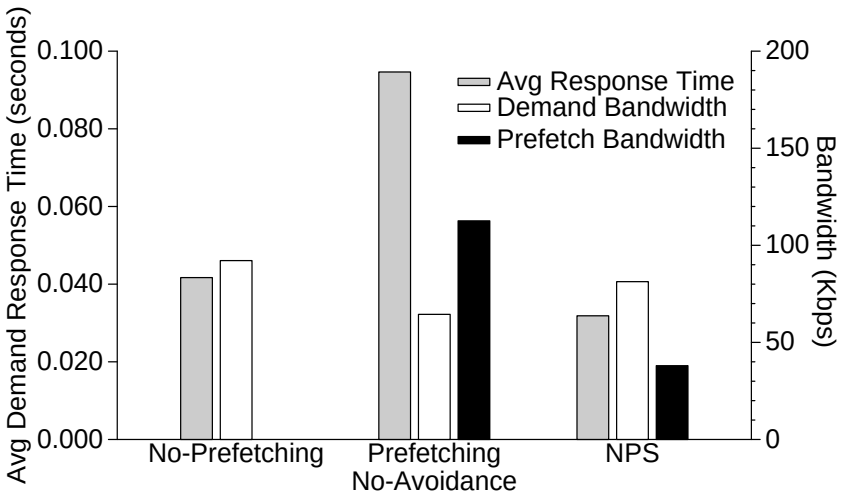
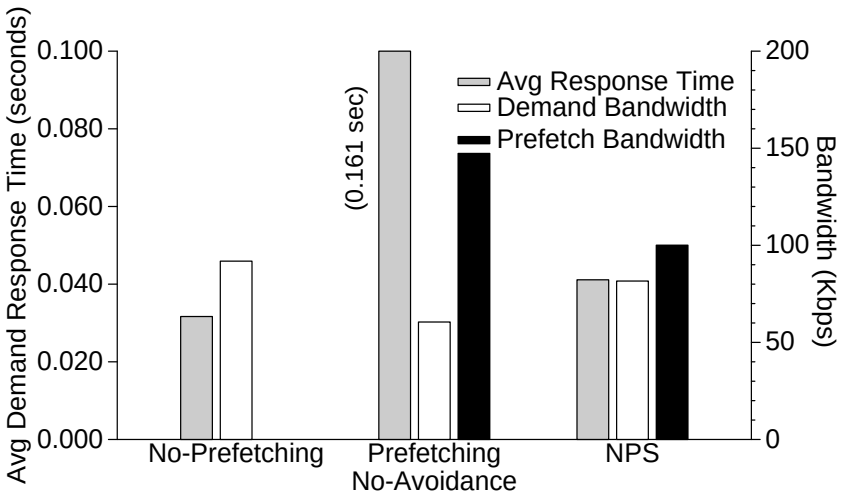



Figure 4.19: Effect of prefetching on demand response times with a loaded server

We use httperf to replay a subset of the IBM server trace. The trace is one hour long and consists of demand accesses made by 42 clients. This workload contains a total of 14044 file accesses of which 7069 are unique; the demand network bandwidth is about 92 Kbps. We modify httperf to simulate the execution of JavaScript as shown in Figures 4.14, 4.15 and 4.16. Also, we modify httperf to implement a large cache per client that never evicts a file that is fetched or prefetched during a run of an experiment. In No-Avoidance case, we set the pfrate to 70, i.e. it gets a list of 70 files to prefetch, fetches them and stops. This pfrate is such that neither the server nor the network becomes a bottleneck even for the No-Avoidance case. For NPS, we assume that each document will consist of ten files (a document is a HTML page along with the embedded objects). Thus the hint server gives out hint lists of size 10 to the requesting clients. Note that many of the files given as hints could be cache hits at the client.



Effect of prefetching on demand response times with a loaded network

**Unloaded resources** In this experiment, we use the setup explained above. Figure 4.18 shows that when the resources are abundant, both No-Avoidance and NPS cases significantly reduce the average response times by prefetching. The graph also shows the bandwidth achieved by No-Avoidance and Nice.

**Loaded server** This experiment demonstrates the effectiveness of the monitor as an important component of NPS. To create a loaded server condition, we use a client machine connected on a LAN to the server running httperf that replays a heavier subset of the IBM trace and also prefetches like the WAN client. Figure 4.19 plots the average demand response times and the bandwidth used in the three cases. As expected, even though the server is loaded, the clients prefetch aggressively in the No-Avoidance case, thus causing the demand response times to increase by more than a factor of 2 rather than decrease. NPS, being controlled by the monitor, prefetches less data and hence avoids any damage to the demand response times. NPS in fact benefits from prefetching, as shown by the decrease in the average demand response time.

**Loaded network** This experiment demonstrates the effectiveness of TCP-Nice as a building block of NPS. In order to create a heavily loaded network with little spare capacity, we set up another client machine running httperf that shares the cable modem connection with the original client machine, replays the same trace, and also prefetches like the original client. Figure 4.3.8 plots the average demand response times, demand bandwidth, and prefetch bandwidth in all three cases. The results show that when the network is loaded, No-Avoidance causes significant interference to demand requests, thereby increasing the average demand response times by a factor of 7. Although NPS doesn't show any improvements, it contains the increase in demand response times to less than 30%, which shows the effectiveness of TCP-Nice in avoiding network interference. The damage is because TCP-Nice is primarily designed for long flows.

### 4.3.9 Related Work

Several studies have published promising results that suggest that prefetching (or pushing) content could significantly improve web cache hit rates by reducing compulsory and consistency misses [50, 64, 86, 89, 111, 114, 140, 174]. However, existing systems either suffer from a lack of deployability or use threshold-based magic numbers to address the problem of interference. Several existing commercial client-side prefetching agents that require new code to be deployed to clients are available [134, 98, 177]. At least one system makes use of Java applets to avoid modifying browsers [73]. It is not clear however, what, if any, techniques are used by these systems to avoid self- and cross-interference.

Duchamp [64] proposes a fixed bandwidth limit for prefetching data. Markatos [129] adopts a popularity-based approach where servers forward the $N$ most popular documents to clients. Many of these studies [64, 102, 174] propose prefetching an object

if the probability of its access before it gets modified is higher than a threshold. The primary performance metric in these studies is increase in hit rate. However, the right measures of performance are end-to-end latency when many clients are actively prefetching, and interference to other applications.

Davison et. al [60] propose using a connectionless transport protocol and using low priority datagrams (the infrastructure for which is assumed) to reduce network interference. Servers speculatively push documents chunked into datagrams of equal size and (modified) clients use range requests as defined in HTTP/1.1 for missing portions of the document. Servers maintain state information for prefetching clients and use coarse-grained estimates of per-client bandwidth to limit the rate at which data is pushed to the client. Their simulation experiments do not explicitly quantify interference and use lightly loaded servers in which only a small fraction of clients are prefetching. Crovella et. al [50] show that a window-based rate controlling strategy for sending prefetched data leads to less bursty traffic and smaller queue lengths.

In the context of hardware prefetching, Lin et. al [122] propose issuing prefetch requests only when bus channels are idle and giving them low replacement priorities so as to not degrade the performance of regular memory accesses and avoid cache pollution. Several algorithms for balancing prefetch and demand use of memory and storage system have been proposed [34, 38, 107, 143]. Unfortunately, applying any of these schemes in the context of Web prefetching would require modification of existing clients.

## 4.4 Providing Consistency Guarantees in Mars

Mars' architecture as introduced in Section 4.2, and the sample application of a Web prefetching system, NPS, did not address the issue of maintaining consistency requirements imposed by applications. It is easy to that ASR can improve response times and availability of large-scale replicated systems that require weak or no consistency semantics. It is less obvious that these improvements extend to applications with strict consistency requirements. In this section, we show how to augment Mars' architecture in order to provide arbitrary levels of consistency guarantees in ASR-enabled systems. We then illustrate through examples how to model practical systems using this augmented architecture.

The key to understanding how to provide consistency in Mars-based ASR systems is the observation that consistency information is meta-information that can be propagated independently of the data or update that it is associated with. An example of consistency information is an *invalidate* message that can be sent to a cache to inform it that the copy it is caching is stale. This invalidate can be sent independently of the actual updated data itself that may arrive via a separate channel. Separation of consistency or invalidate information from updates thus enables Mars-based ASR systems to maintain consistency semantics desired by an application. The architecture illustrated in Figure 4.2 can be used to propagate consistency information via the demand channel at regular priority while speculative updates arrive via the background channel. The system can continue to use legacy protocols for maintaining consistency with minor modifications, i.e. by replacing whole updates with small invalidates of a fixed size. As before, updates, like speculatively replicated objects, are simply maintained and propagates in priority order to locations where they are expected to be accessed.

Note that consistency information and demand requests do not have to share a common channel. For example, for some applications such as distributed financial

transactions, it might be important to transmit invalidate information with timeliness guarantees without interference from even demand requests. In such cases, consistency information may be propagated using an even higher priority channel, or an entirely different network providing enhanced quality of service guarantees. Demand and speculative update bodies can be propagated as in the original Mars architecture. Separation of consistency and updates is key for this kind of flexibility. Note that Mars' architecture itself does not enforce the need for a channel for consistency information that is different from that used to propagate demand requests. Legacy applications need only modify consistency maintenance protocols to work with invalidates as opposed to whole update bodies. The point here is that if the the application already has a separate enhanced propagation infrastructure available for consistency information, then it can be smoothly integrated with Mars' architecture.

We remark that this augmented architecture of Mars does not let us circumvent the CAP impossibility result introduced in Chapter 1. For example, if an application receives an invalidate but not the corresponding update for an object, then a request for that object cannot be satisfied. In some applications, requiring linearizablity for instance, a disconnected replica cannot be allowed to serve a cached copy even if it has not received any invalidate for the copy. Though Mars does not help the CAP result for the general replication problem, it does allow for improved availability for several classes of applications through its support for ASR. For example, if an application seeks only $\Delta$-consistency (i.e., only copies no staler than a time $\Delta$ may be served), an ASR-enabled system is likelier to have a fresh enough copy than a system that does not use ASR. An example of a system that seeks to provide $\Delta$-consistency is a commercial content distribution system like Akamai [5].

We give below two examples of systems requiring consistency guarantees that can leverage Mars' architecture for ASR. The first is that of a data dissemination

service described in detail in the work by Nayate et al. [135], and the second is that of a replicated file system like Bayou described in detail in the work by Dahlin et al. [56]. We show how these systems can be redesigned along Mars' architecture and enabled to use ASR.

**A Dissemination Service** We consider a data dissemination service consisting of a primary server and a collection of replica servers distributed across a WAN. Clients make read requests at replica locations. Writes, i.e. updates to existing objects and creation of new objects, happens only at the primary. The consistency semantics demanded by the service is that of sequential consistency. Clearly, such a system can benefit from speculative propagation of writes to replica locations. Sequential consistency is simply maintained by associating each write with a sequence number that represents the order in which the write was committed at the primary. Replicas process both speculative and demand writes in the common order dictated by the sequence number. Nayate et al. [135] show that sequential consistency is indeed maintained because of first-in-first-out (FIFO) property of global sequence numbers.

However, the simple system as described above cannot perform ASR as demand response times shoot up drastically due to interference and network overload. However, the system be redesigned using Mars to enable ASR as follows. The primary separates the traffic it sends to replicas into three categories - i) demand requests, ii) invalidate information, and iii) speculative update bodies. Consistency information can be propagated on the same channel as that used for demand requests while speculative update bodies arrive via a background channel in priority order. Each replica continues to apply all messages in the common sequence number order. This modified system can yield significant reductions in both response times, availability and bandwidth consumption at replicas. Response times get reduced because available bandwidth is used to speculatively replicate in the background preventing interference and system overload. Bandiwdth usage is reduced as only those updates

are speculatively replicated whose likelihood of being used is high. ASR improves availability as a disconnected replica can continue serving cached copies for a longer duration. We omit details of certain workarounds needed to continue to maintain sequential consistency in this modified system and refer the interested reader to Nayate et al. [135] for the detailed implementation and evaluation of such a system.

**A Replicated File System** We consider a sample replicated file system like Bayou [168]. Bayou consists of a collection of replicas all of which maintain a complete replica of a database. Writes and reads to objects in the database may occur at each replica and the system strives to maintain causal consistency of views of the database seen by a replica. Bayou functions through an *anti-entropy* protocol wherein a replica can exchange updates with any other replica to bring each other closer to the eventual committed version of the database. Replicas maintain the causality property by using logical Lamport clocks [116] for local writes and a version vector that represents its view of what updates other replicas have seen.

Unfortunately, the Bayou anti-entropy protocol ends up transmitting complete bodies of all updates that the other replica has not yet seen. This protocol incurs heavy bandwidth overheads as all updates are propagated to all replicas. and thus may be impractical to use in bandwidth-constrained environments. Moreover, though causal consistency is maintained, a replica may serve very stale data as bandwidth constraints may prevent a replica from receiving a fresher update for a long time. As an alternative, this system may be redesigned using the Mars approach as follows. As in the above example, each replica separates its exchanges with other replicas into three categories - i) demand requests, i) invalidates, iii) update bodies. Notice that in the original Bayou system, demand requests could always be served locally. However, by separating invalidates and updates, a replica may need to contact other replicas for a causally consistent version of the requested object. The anti-entropy protocol is simply modified to work with invalidates instead of

whole update bodies, while update bodies are propagated in a prioritized manner, like in Mars, to only those replicas that need the updates. Thus, Bayou redesigned using Mars' architecture augmented with separation of consistency and update information can yield significant reductions bandwidth usage. Dahlin et al. [56] give a detailed description of this redesign and a rigorous evaluation of the reductions in bandwidth consumption.

# Chapter 5

# Bandwidth-Constrained Speculative Relication

In the previous sections, we introduced mechanisms for aggressive speculative replication. The research methodology was to develop mechanisms and validate them through analytical means as well as by building prototypes to understand system properties. In the next three chapters, we focus on policy issues related to ASR. The research methodology is primarily based on proving system properties in a simplistic analytical framework and on simulation based experiments.

This chapter examines the problem of choosing what objects to speculatively replicate, or prefetch, at a large content distribution site. Towards this end, we study the costs and potential benefits of a technique called *long-term prefetching* for content distribution. In contrast with traditional short-term prefetching, in which caches use recent access history to predict and prefetch objects likely to be referenced in the near future, long-term prefetching uses long-term steady-state object access rates and update frequencies to identify objects to replicate to content distribution locations. Compared to demand caching, long-term prefetching increases network bandwidth and disk space costs but may benefit a system by im-

proving hit rates. The techniques of short-term prefetching, long-term prefetching, and demand caching are complementary and we envision systems that incorporate all three techniques for improving response times.

We use analytic models and trace-based simulations to examine several algorithms for selecting objects for long-term prefetching. We find that although the web's Zipf-like object popularities makes it challenging to prefetch enough objects to significantly improve hit rates, systems can achieve significant benefits at modest costs by focusing their attention on long-lived objects.

## 5.1 Introduction

In spite of advances in web proxy caching techniques in the past few years, proxy cache hit rates have not improved much. Even with unlimited cache space, passive caching suffers from uncacheable data, consistency misses for cached data and compulsory misses for new data. Prefetching attempts to overcome these limitations of passive caching by proactively fetching content without waiting for client requests. Traditional short-term prefetching at clients uses recent access history to predict and prefetch objects likely to be referenced in the near future and can considerably improve hit rates [28, 30, 64, 70, 142].

In this chapter, we examine a technique more appropriate for large proxies and content distribution networks (CDNs), namely *long-term prefetching*. Rather than basing prefetching decisions on the recent history of individual clients, long term prefetching seeks to increase hit rates by using global object access patterns to identify a collection of valuable objects to replicate to caches and content distribution servers.

As hardware costs fall, more aggressive prefetching becomes attractive making it possible to store an enormous collection of data at a large content distribution site. For example, in March 2001 an 80GB disk drive cost about $250 [57]. How-

ever, maintaining a collection of hundreds of gigabytes or several terabytes of useful web data incurs not just a space cost but also a bandwidth cost: as objects in the collection change, the system must fetch their new versions. It must also fetch newly created objects that meet its selection criteria. Due to the Web's Zipf-like access patterns, a large number of objects must be actively prefetched to improve hit rates significantly [28]. Maintaining such a collection appears to be challenging. In particular, bandwidth expenditure will be the primary constraint in a long term prefetching strategy. For example, in May 2001 a 1.5 Mbps T1 connection cost about $1000 per month [100].

In this chapter, we present a model for understanding steady-state cache behavior in a bandwidth-constrained prefetching environment. Our hypothesis is that by prefetching objects that are both long-lived and popular, we can significantly improve hit rates for moderate bandwidth costs. The key contribution of our work is a Goodfetch algorithm for long term prefetching that balances object access frequency and object update frequency and that only fetches objects whose probability of being accessed before being updated exceeds a specified threshold determined by the bandwidth limit.

Using synthetic and real proxy trace based simulations we establish that our algorithm provides significant hit rate improvements at moderate storage and bandwidth costs. For example for a modest-size cache that receives 10 demand requests per second, long-term prefetching can improve steady state hit rates for cacheable data from about 62% (for an infinite demand-only cache) to above 75% while increasing the bandwidth demands of the system by less than a factor of 2. More generally, we quantify the trade-offs involved in choosing a reasonable prefetch threshold for a given object access rate. Based on our trace based simulation, we conclude that the key challenge to deploying such algorithms is developing good predictors of global access patterns. Although we leave development of such predictors as future work,

we provide initial evidence that even simple predictors may work well.

The rest of this chapter is organized as follows. Section 5.2 provides some background information about prefetching and our prefetching model. Section 5.3 presents the algorithms that we consider for long-term prefetching. Section 5.4 discusses the methodology we use to evaluate long-term prefetching. Section 5.5 discusses the results of our simulations and provides insights about how long-term prefetching works. Section 5.6 discusses related work. Section 5.7 summarizes our conclusions.

## 5.2 Background

This section describes five key parameters of web workloads that determine the effectiveness of caching and prefetching: prefetching models, object popularities, object sizes, object update patterns and lifetimes, and the availability of spare bandwidth for prefetching.

### 5.2.1 Prefetching Models

We categorize prefetching schemes into two groups: short-term and long-term. In the short-term model, a cache's recent requests are observed and likely near-term future requests are predicted. Based on these predictions, the objects are prefetched. Considerable research has been performed on this type of model [64, 70, 142], most of which are based on variations of a Prediction-by-Partial-Matching (PPM) strategy [53].

In the long-term model of prefetching on which we focus, we assume that a cache or content distribution site maintains a collection of replicated objects based on global access pattern statistics such as object popularity and update rates. We envision a hierarchical structure for content distribution with lower level caches (proxy caches) primarily focusing on servicing client requests and the higher level

caches (content distribution servers) on effective content distribution using long-term prefetching. Proxy caches can use short term prefetching to improve hit rates further. Content servers maintain a collection of popular objects and update these objects as they change. New objects are added to the collection based on server assistance and user access.

The content distribution system requires four components:

1. *Statistics tracking.* Our selection algorithm uses as input: (i) estimates of object lifetimes and (ii) estimates of access frequency to objects. Maintaining these estimates is a key challenge to deploying a long-term prefetching based system, and we do not address this problem in detail.

   If content servers are trusted by the content distribution system, they may be able to provide good estimates. Otherwise, the system itself must gather access probability reports from clients or caches and track object update rates. For example, a distributed federation of caches and content distribution nodes could gather local object access distributions and report these statistics to a central aggregation site which would distribute the aggregate statistics to the caches and nodes. There is some evidence that relatively short windows of time can provide good access estimates [104].

2. *Selection criteria.* Based on the statistics, the selection criteria module determines which objects should be included in the replica's collection. The rest of this section discusses this issue in detail.

3. *Data and update distribution.* The system must distribute objects and updates to objects to caches that include the objects in their collection of replicated objects. We model a push-based system in which updates to replicas are sent immediately to caches that have "subscribed" to the object in question. In this work, we do not address the details of constructing such a push-based

system and refer the interested reader to relevant literature [135, 89].

4. *Request redirection.* In order to enable clients to transparently access a nearby copy of a replicated object, an effective redirection scheme is needed. A number of experimental [69, 190] and commercial systems [4, 101] address this issue.

### 5.2.2 Popularity Distributions

A key parameter for understanding long-term prefetching is the distribution of requests across objects. Several studies [6, 52, 78, 181] have found that the relative distribution with which Web pages are accessed follows a Zipf-like distribution. Zipf's law states that the relative probability of a request for the $i$'th most popular object page is inversely proportional to $i$. Cunha et al. [52] found that the request probability for a Web cache trace, when fitted with a curve of the form $1/i^{\alpha}$, yields a curve with exponent of $\alpha = 0.982$ which we will use as a default parameter. Other researchers have reached similar conclusions [28].

According to this model, given an universe of $N$ Web pages, the relative probability of $i$th most popular page is

$$p_i = \frac{C}{i^{\alpha}}, \text{ where } C = \frac{1}{\sum_{k=0}^{N} \left(\frac{1}{k^{\alpha}}\right)} \tag{5.1}$$

For our synthetic workload, we will use this model of accesses with $N = 10^9$, $\alpha = .982$, and $C = 0.0389$.

### 5.2.3 Object sizes

Studies by Barford and Crovella [50] show that web object sizes exhibit a distribution that is a hybrid of a log-normal and a heavy tailed Pareto distribution. The average size of a web object has been shown to be around 13KB. Work by Breslau et al. [28] suggests that there is little or no correlation between object sizes and their popularity. However, an earlier study by Crovella et al. [52] claims an inverse

relationship between object sizes and popularities, *i.e.* users *prefer* small documents. They show a weak Zipf correlation between popularity and size with a zipf parameter -0.33. However we did not observe an appreciable correlation between object sizes and popularity in the Squid traces we analyzed. Hence, for our simulations we do not assume any correlation. It must, however, be emphasized that if the inverse correlation were to be assumed, a prefetching strategy based on maintaining popular objects will perform better with respect to both bandwidth and cache size.

### 5.2.4 Update patterns and lifetimes

Web objects have two sources of change - (i) updates to objects that are already present, (ii) introduction of new objects. The work by Douglis et al. [62] shows that (mean) lifetimes of web objects are distributed with an overall mean of about 1.8 months for html files and 3.8 months for image files. Though they analyzed lifetimes for objects in varying popularity classes, little correlation is observed between lifetime and popularity. The work by Breslau et al. [28] further strengthens the case for lack of strong correlation between lifetime, popularity and size of objects.

The lifetime distribution for a single object over time is found to be exponential in [30]. They consider the Internet as an exponentially growing universe of objects with each object changing at time intervals determined by an exponential distribution. Their analysis shows that the age distribution of an exponentially growing population of objects with (identical) exponential age distributions remains exponential with the parameter given by the sum of the population growth and object update rate constants. They then show with respect to the cost of maintaining a collection of fresh popular objects that the introduction of new objects on the Internet is equivalent to changing objects in a static universe of objects with a different rate parameter.

In our simulations we use the data for lifetime distribution presented in [62].

We assume no correlation with popularity or size. Our criterion for selecting an object is based on its current popularity and mean lifetime and is independent of its past and of other objects. We therefore describe our algorithm in terms of the bandwidth cost to update a fixed collection of objects as they are updated. But following analysis done by Brewington et al. [30], our algorithm and analysis also apply to the case of maintaining a changing collection of objects that meet the system's replication selection criteria. We explain this assumption in greater detail in section 5.4.

### 5.2.5 Spare Prefetch Resources

Prefetching increases system resource demands in order to improve response time. This increase arises because not all objects prefetched end up being used. Resources consumed by prefetching include server CPU cycles, server disk I/O's, and network bandwidth. The increased resource consumption can interfere with demand requests. Interference can increase miss latencies for demand requests and offset the benefits of prefetching as explained in Chapter 4. Excessive prefetching can also overload systems causing severe performance degradation compared to even a system that does no prefetching. Therefore, a key issue in understanding prefetching is to determine an appropriate balance between increased resource consumption and improved response time.

### 5.2.6 Methodology

In Chapters 1-4, we addressed the mechanism issue of how to achieve this balance in a self-tuning manner to support aggressive prefetching. In this chapter, we explore the policy question of what objects to aggressively prefetch given bandwidth constraints at a large content distribution site. In order to examine the effect of a particular level of aggressiveness, we simulate it using a *prefetch threshold*, i.e. a file is prefetched

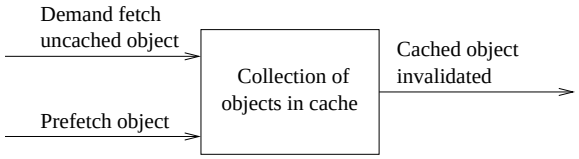



Figure 5.1: Equilibrium in bandwidth-constrained cache.

if the probability of its access is above the threshold. Note that the purpose of the threshold in the following experiments is strictly as an aggressiveness metric and is used to determine the increased bandwidth and storage costs of prefetching and the associated benefits in response times assuming that there is no interference between prefetch and demand requests. In a real system employing ASR, Mars' self-tuning architecture will obviate the need for explicitly setting thresholds and will automatically propagate prefetch requests commensurate to available spare capacity without interfering with demand requests.

## 5.3 Model and algorithms

In contrast with traditional caching, where space is the primary limiting factor, for long-term prefetching bandwidth is likely to be the primary limiting factor. In this section, we first describe an equilibrium model useful for understanding the dynamics of bandwidth-constrained long-term prefetching. We then describe our algorithms.

### 5.3.1 Bandwidth Equilibrium

A long-term prefetching system attempts to maintain a collection of object replicas as these objects change and new objects are introduced into the system.

Figure 5.1 illustrates the forces that drive the collection of fresh objects stored in a cache towards equilibrium. New objects are inserted into the cache by demand requests that miss in the cache and by prefetches. Objects are removed from the set

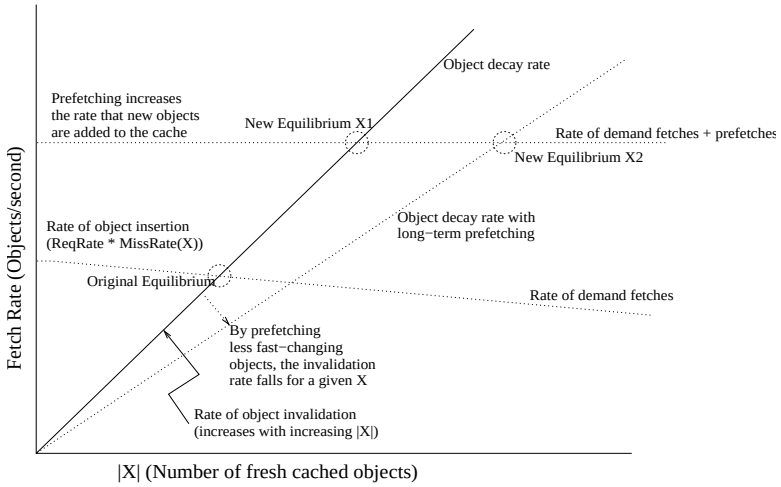



Figure 5.2: Equilibrium in bandwidth-constrained cache.

of fresh objects in the cache when servers update cached objects, invalidating the cached copy.[1]

The solid lines in Figure 5.2 illustrates how this equilibrium is attained for a demand-only cache with no prefetching. Let $X$ be the set of fresh objects in the cache at a given moment, and let $|X|$ denote the number of objects in this set. If the number of requests per second being sent to the cache is $ReqRate$, then the rate of object insertion into this set is $ReqRate \cdot MissRate(X)$. For a given request rate, the miss rate typically falls slowly as $|X|$ increases [65, 84], and the rate of insertion falls with it. At the same time the rate of invalidations (or expirations) of cached objects increases as $|X|$ increases. As the figure illustrates, these factors combine to yield an equilibrium collection of objects that can be maintained in the cache.

The dotted lines in Figure 5.2 illustrate how prefetching can change this equilibrium. First, as the horizontal line illustrates, prefetching increases the rate at which new objects are added to $X$. If the collection of objects prefetched have

[1]For simplicity, we describe a system in which servers invalidate clients' cached objects when they are updated [49, 113, 123, 185]. Client-polling consistency would yield essentially the same model: in that case, objects that expire are removed from the set of objects that may be accessed without contacting the server.

similar lifetimes to the collection of objects fetched on demand, then invalidation rates will behave in a similar fashion, and a new equilibrium with a larger set $X$ will be attained as shown by the point labeled *New Equilibrium* $X_1$.

A prefetching system, however, has another degree of freedom: it can choose what objects to prefetch. If a prefetching system chooses to prefetch relatively long-lived objects, its invalidation rate for a given number of prefetched objects $|X|$ may be smaller than the invalidation rate for the same number of demand fetched objects. This change has the effect of shifting the invalidation rate line down, and yields a new equilibrium, *New Equilibrium* $X_2$, with $|X_2| > |X_1|$.

A potential disadvantage of preferentially prefetching long-lived objects is that the system may thereby reduce the number of frequently-referenced objects it prefetches. In particular, although $|X_2| > |X_1|$, if the objects in $X_1$ are more popular than the objects in $X_2$, the hit rate for equilibrium $X_1$ may exceed the hit rate for equilibrium $X_2$.

### 5.3.2 Prefetching Algorithms

We consider two prefetching algorithms. The first attempts to maintain an equilibrium cache contents containing the most popular objects in the system. The second attempts to balance object popularity (which represents the benefit of keeping an object) against object update rate (which represents the cost of keeping an object.)

#### Popularity

The Popularity algorithm identifies the $k$ most popular objects in the universe and maintains copies of them in the cache. Whenever any one of these objects is updated (or a new object joins the set of the most popular $k$ objects), the system fetches the new object into the cache immediately.

Figures 5.3 and 5.4 shows the hit rate achieved and bandwidth consumed by

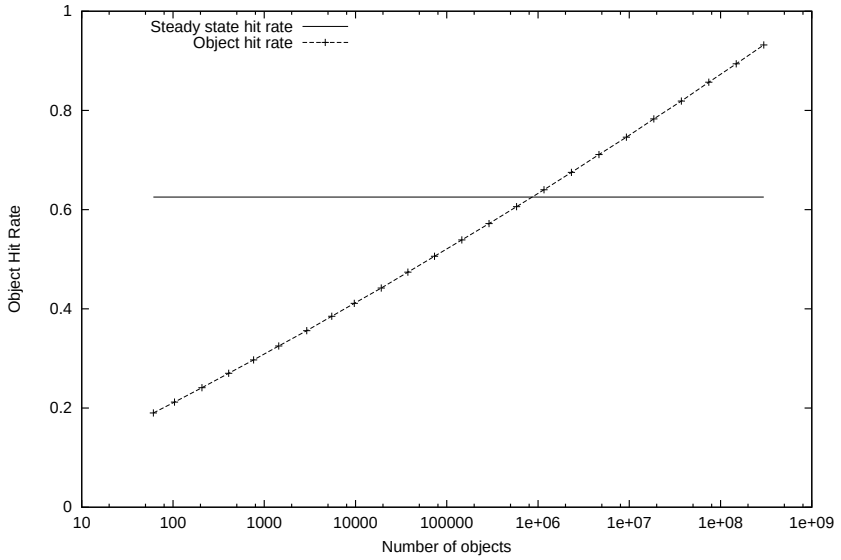



Figure 5.3: Hit rate vs. number of prefetched objects for the Popularity algorithm

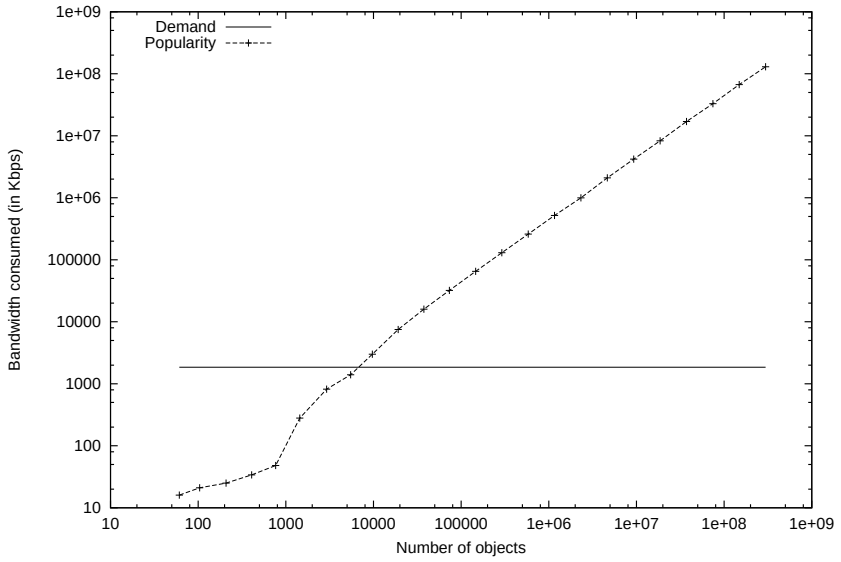



Figure 5.4: Bandwidth vs. number of prefetched objects for the Popularity

the Popularity algorithm when it prefetches the $k$ most popular objects. Although this approach can achieve high hit rates, its bandwidth costs are high. In particular, to improve the hit rate by 10% compared to the steady state hit rate of a demand cache that receives 10 requests per second, the Popularity algorithm must consume at least 1000 times more bandwidth than such a demand cache.

Because of these high bandwidth demands, we do not consider the Popularity algorithm further in this section.

### Goodfetch

The Goodfetch algorithm balances object access frequency and object update frequency and only fetches objects whose probability of being accessed before being updated exceeds a specified threshold. In particular, for object $i$, given the object's lifetime $lifetime_i$, the probability that a request will access that object $P_i$, and the total request rate of demand requests to the cache $requestRate$ and assuming that object references and updates are independent, the probability that a cache will access an object that it prefetches before that object dies is

$$Goodfetch = 1 - (1 - P_i)^{lifetime_i \cdot requestRate} \tag{5.2}$$

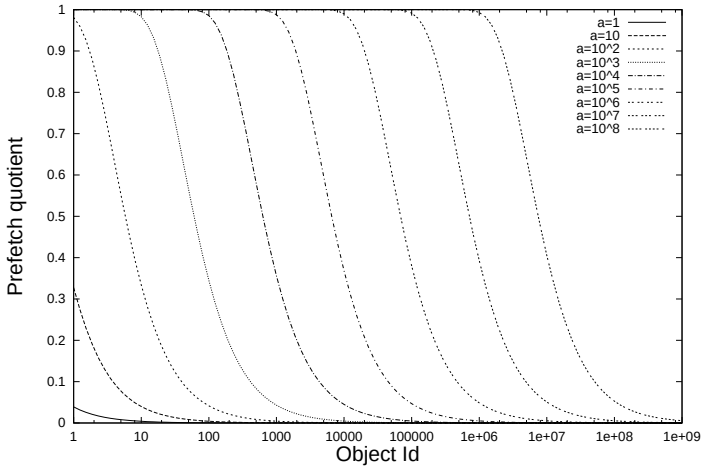
$lifetime_i * requestRate$ is the total number of requests to the cache expected during the object lifetime, and $(1-P_i)^{lifetime_i \cdot requestRate}$ is the probability that none of these requests access object $i$.

The Goodfetch values computed as above are used to prioritize objects to be prefetched and maintain them as a sorted list. As illustrated in Figure 4.2, the most valuable object in this list is pushed out when spare capacity is available. In our simulation experiments, however, we simply use a threshold to limit the total bandwidth consumed by prefetching and do not simulate interference effects. As demonstrated through NPS in Chapter 4.3, simple end-to-end mechanisms may be

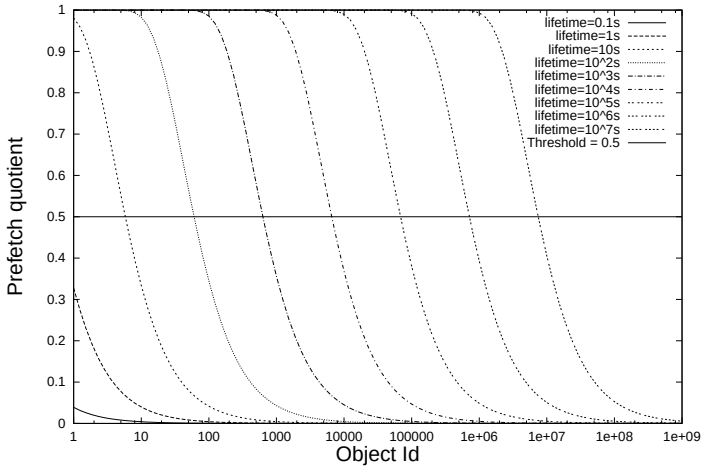
used to prevent such interference. In the following experiments, the threshold is used to limit prefetch bandwidth by prefetchng only those objects whose Goodfetch value exceeds the threshold corresponding to the available prefetch bandwidth.

Figure 5.5-a shows a "contour map" that relates the Goodfetchvalues to a Zipf-like popularity distribution. Objects are sorted by popularity or access frequency and the $x$-axis represents the object index $i$ in this sorted order. Assuming accesses to objects follows a Zipf-like distribution $P_i \approx \frac{C}{i^\alpha}$, $P_i$ falls slowly with increasing $i$. Each line in the figure represents a line of constant $objectLifetime \cdot cacheRequestRate$, and the $y$-axis represents corresponding Goodfetch values. $ObjectLifetime$ depends on the particular object chosen and $cacheRequestRate$ depends on the demand fetch rate to the cache under consideration. A given value of threshold in these experiments corresponds to a particular amount of spare bandwdith available for prefetching in the system. Given a threshold for a cache, the set of objects that qualify for prefetching based on their access frequencies and update rates is determined. Figure 5.5-b illustrates the set of objects that will be prefetched for a threshold of 0.5 and a request rate of 10 request/second. Objects among the top-1000 are worth prefetching if they will live at least $10^3$ seconds (about 17 minutes); in contrast objects among the top-1,000,000 are worth preserving if they will live at least $10^6$ seconds (about 12 days). Given that a significant fraction of objects live for 30 days or longer [62], a busy cache such as this may be justified in prefetching a large collection of objects.

Overall, it appears that long-term bandwidth-constrained prefetching will be most attractive for relatively large, busy caches. Reducing the request rate to a cache by an order of magnitude reduces the set of objects eligible to be prefetched to that cache for a given threshold by approximately an order of magnitude, i.e. as the access rate at a cache decreases, the number of objects that can be prefetched at that cache for a given amount of spare bandwidth decreases considerably.




(a) Goodfetch map




(b) Example

Figure 5.5: Prefetch threshold vs object popularity for lines of constant $objectLifetime \cdot cacheRequestRate$. (a) shows the threshold map and (b) illustrates the set of objects that will be prefetched for a threshold of 0.5 and a request rate of 10 request/second.

### 5.3.3 Constant-factor Optimality

In this section we show that the hit rate obtained by prefetching the set of objects computed by the Goodfetch algorithm is within a constant factor (2X) of the optimal achievable hit rate under a simple restrictive model. The model assumes that the cache gets a long sequence of requests so that average access rates and average lifetimes of objects can be computed and are known a priori. It also assumes that requests for objects arrive independently in accordance with the respective access rates. We show the optimality to within a constant factor by expressing the object selection problem in the context of prefetching as equivalent to the 0-1 Knapsack problem. We then show that the Goodfetch algorithm is equivalent to the *value-density* heuristic for the 0-1 Knapsack problem, which is known to be within a 2X factor of optimal solution, but for a technical condition described below.

We first introduce the Knapsack problem which is as follows: Given a set of items $S = \{1, \ldots, n\}$, where item $i$ has size $s_i$ and value $v_i$ and a knapsack with capacity $C$, find the subset $S' \subseteq S$ that maximizes the value of $\sum_{i \in S'} v_i$ given that $\sum_{i \in S'} s_i \leq C$.

The value-density heuristic for the Knapsack problem is as follows: Let $d_1, d_2, \ldots, d_n$ denote the items in the order of non-increasing value-density, *i.e.* $\frac{v_i}{s_i}$. Return the set of objects as $S' = \{d_1, \ldots, d_k\}$, where $k$ is the largest integer such that $\sum_{1 \leq i \leq k} d_i \leq C$. The following lemma expresses the optimality of the value-density heuristic to within a constant factor provided the following technical condition is satisfied: no single object has greater value than the combined value of the set of objects computed by the value-density heuristic.

*Lemma 1*: Let $P_{opt}$ be the value of the optimal selection of objects and $P$ denote the value of the set computed by the value-density heuristic. Then, $P_{opt} \leq 2(max(P, max_{i \in [1,n]}(v_i)))$

**Theorem 1**: The byte hit rate achieved by the Goodfetch algorithm is within

a 2X factor of the optimal.

*Proof Outline*: We model the object selection problem in the context of prefetching as an instance of the $0-1$ Knapsack problem. Given a set of $n$ objects, with a popularity distribution $P_i$, a lifetime distribution $lf_i$, $1 \leq i \leq n$, a size distribution $sz_i$, a total bandwidth constraint $B$ and an infinite demand cache, the prefetching problem is to select a suitable subset of objects $S$ to keep updated locally in order to maximize hit rate. The contribution of object $i$ to the object hit rate is $P_i$. However, the contribution to the hit rate because of prefetching object $i$, given the existence of an infinite demand cache, is the probability that object $i$ is accessed at least once during its lifetime, divided by $l_i$ which is given by $\frac{Goodfetch(i)}{l_i}$. The contribution to the byte hit rate is therefore proportional to $\frac{Goodfetch(i)*sz_i}{l_i}$. We understand this expression as the *value* of object $i$. The "size" or the bandwidth cost of keeping the local copy of object $i$ updated is $\frac{sz_i}{lf_i}$. Therefore, the *value-density* of object $i$ is given by $Goodfetch(i)$, which is exactly the criterion that the Goodfetch algorithm uses to select objects. Thus the Goodfetch algorithm is equivalent to the value-density heuristic for the 0-1 Knapsack problem and hence achieves a byte hit rate that is within a 2X factor of the optimal provided that the following technical condition is satisfied: no single object $i$ has a value, i.e. $\frac{Goodfetch}{l_i}$, the contribution to the hit rate because of prefetching $i$, greater than the combined value of the set of objects selected by the Goodfetch critrerion. □

## 5.4 Methodology

In the previous section we introduced the Goodfetch algorithm that computes a collection of objects for which the probability of access before it gets updated is maximal. We claimed that this algorithm balances object access frequency and update frequency thereby resulting in minimal wasted prefetch bandwidth and still gives attractive improvements in hit rate. In order to verify this, we perform a cost-

benefit analysis of the Goodfetch algorithm in terms of improvement in hit rate, bandwidth consumption and cache size through two sets of simulation experiments - (i) based on a synthetically generated set of 1 billion objects, and (ii) a proxy trace based simulator implementing a prediction based version of the Goodfetchalgorithm. The synthetically generated workload experiments fundamentally give a proof of concept for the performance of the Goodfetch algorithm. A key benefit of using a synthetic workload is that it allows us to model global object popularities, including objects that have not been accessed in the trace of a particular cache. On the other hand the proxy trace based experiments analyze the performance of the implementation of an adaptive, prediction based version of the Goodfetch algorithm on a smaller but realistic workload. This workload automatically exhibits temporal locality between accesses to the same object, their size distribution and models burstiness in request traffic as well. It is directly comparable to the performance of real web proxies and hence serves as a sanity check.

### 5.4.1 Analytic model

For simplicity, we assume the demand request arrival rate follow a Poisson distribution (We discuss the effect of this assumption later in the section). For calculation of steady state hit rates, we assume that the request arrival rate has a mean of $a$ requests per second. We define the lifetime of an object as the time between two modifications to that object. The lifetimes of an object are known to follow an exponential distribution [30]. In our model, we assume that the average lifetime of an object $i$ to be $l_i$. Each object $i$ is assumed to have a fixed size $s_i$. As previously explained, the probability of an access to an object with popularity $i$ follows a zipf-like distribution. We assume the objects are indexed with respect to their popularities. Hence the probability of a request is for an object with index $i$ is $p_i = C * (1/i^{\alpha})$.

### Steady state demand hit rate

In this section we present a closed form expression for calculating the steady state hit rate of a demand cache with an infinite cache size. An analytical expression is necessary as the simulation based study is not plausible for calculating the steady state parameters because of the huge number of objects and hence the long time requirements for cache warm up. We define $P_{A_i}(t)$ as the probability that the previous access to object $i$ was $t$ time units before the present time and $P_{B_i}(t)$ as the probability that no updates were done to object $i$ since its last access $t$ time units in the past. Now, the probability of a hit on a request to a demand cache is

$$P_{hit_d} = \sum_i p_i \int_0^\infty P_{A_i}(t) P_{B_i}(t) dt, \tag{5.3}$$

Suppose $P_{(a,t)}(k)$ is the probability of $k$ accesses occurring in $t$ time given an access arrival rate of $a$. With the assumption of request arrivals following Poisson distribution, the probability of $k$ arrivals occurring in $t$ seconds is $P_{(a,t)}(k) = e^{-at}.\frac{(at)^k}{k!}$. The probability of object $i$ being accessed on a request is $p_i$. The probability of no accesses to object $i$ in this time $t$ is

$$\begin{aligned}
P(0 \text{ accesses to object } i \text{ in } t \text{ time}) &= \sum_{k=0}^{\infty} P(k \text{ requests in } t) P(\text{none of these } k \text{ requests is for } i) \\
&= \sum_{k=0}^{\infty} \left( e^{-at} \frac{(at)^k}{k!} \right) (1-p_i)^k, \\
&= e^{-at} \sum_{k=0}^{\infty} \frac{(at(1-p_i))^k}{k!} \\
&= e^{-at} e^{at(1-p_i)} \\
&= e^{-(ap_i)t}
\end{aligned}$$

The above equation implies that the inter access rates to an object $i$ follow an exponential distribution with a mean time of $(1/ap_i)$. Hence, the probability of an access to an object $i$ occurring $t$ time units after an access to the same object is

$(ap_i)e^{-(ap_i)t}$, which is also the distribution for $P_{A_i}(t)$. Hence,

$$P_{A_i}(t) = (ap_i)e^{-(ap_i)t} \tag{5.4}$$

Given that the lifetimes of an object $i$ are exponentially distributed with an average $l_i$, the probability of no updates to that object happen in time $t$ is

$$P_{B_i}(t) = e^{-t/l_i} \tag{5.5}$$

From Equations 5.3, 5.4 and 5.5, the probability of hit on an access will be

$$\begin{aligned} P_{hit_d} &= \sum_i p_i \int_0^\infty \left((ap_i)e^{-(ap_i)t}\right)\left(e^{-t/l_i}\right) dt \\ &= \sum_i p_i(ap_i)\left(\frac{e^{-(ap_i+1/l_i)t}}{-(ap_i+1/l_i)}\Bigg|_0^\infty\right) \\ &= \sum_i p_i\left(\frac{ap_i l_i}{ap_i l_i + 1}\right) \end{aligned} \tag{5.6}$$

The fraction $\frac{ap_i l_i}{ap_i l_i+1}$ represents the hit rate among accesses to the object $i$. Stated otherwise, this denotes the probability of object $i$ being fresh when it is being accessed. We also call this as *freshness factor* of object $i$ denoted as $ff(i)$.

### Steady state prefetch hit rate

In the Goodfetch algorithm we propose for prefetching objects, an object is always kept fresh by prefetching it, when any change occurs, if its $P_{goodFetch}$ as calculated in Equation 5.2 is above chosen threshold value $T$. For an object $i$, if $P_{goodFetch}(i)$ is more than the chosen threshold value $T$, then we will have a hit on that object for all accesses. For other objects the hit rate remains same as calculated in previous which will be $ff(i)$. Hence, the steady state hit rate in a prefetch based scheme

with threshold value $T$ is

$$P_{hit_p}(T) = \sum_i p_i h_i, \text{where} \tag{5.7}$$

$$h_i = \begin{cases} 1 & \text{if } P_{goodFetch}(i) > \text{T} \\ \frac{ap_i l_i}{ap_i l_i + 1} & \text{otherwise} \end{cases}$$

### Steady state cache sizes

The steady state cache size is defined as the total size of fresh objects in a demand cache of infinite size at steady state when the cache is in equilibrium, *i.e.* the rate at new incoming objects are brought into the cache equals the rate at which objects get invalidated. We estimate the steady state demand and prefetch cache sizes in this section. The estimation of these sizes for a billion object real web workload mean a long run of simulation. Assuming a billion object web with 13KB average size, a pessimistic steady state demand cache size is 13TB. However, the steady state cache size as defined above, is likely to be lesser if invalidation of objects were to be taken into consideration.

In steady state, the probability of an object $i$ being fresh in the cache is given by $\int_0^\infty P_{A_i}(t)P_{B_i}(t)dt$, where $P_{A_i}(t)$ and $P_{B_i}$ are as defined earlier. Hence the probability of an object $i$ being fresh in the cache at some random time instance is just the freshness factor of $i$, $ff(i)$ as derived before. Given the freshness factors of objects, the estimated steady state cache size is $\sum_i s_i ff(i)$. Hence, the estimated demand cache size in steady state, $CSize_{ss_d}$, is

$$CSize_{ss_d} = \sum_i s_i \frac{ap_i l}{ap_i l + 1} \tag{5.8}$$

When using the Goodfetch algorithm for prefetching with threshold value $T$, the estimated total cache size is calculated as

$$CSize_{ss_p} = \sum_i s_i * f_i, where \tag{5.9}$$

$$f_i = \begin{cases} 1 & \text{if } P_{goodPrefetch}(i) > \text{T} \\ \frac{ap_i l}{ap_i l+1} & \text{otherwise} \end{cases}$$

**Steady State Bandwidth**

We present the steady state bandwidth requirements for both demand based access methods and the Goodfetchalgorithm based scheme. The estimated steady state bandwidth consumed by just demand fetches is

$$BW_{ss_d} = \sum_i s_i(1 - ff(i))ap_i \tag{5.10}$$

For Goodfetchalgorithm based prefetch strategy, the steady state bandwidth consumed by both prefetch and demand fetches is

$$BW_{ss_p} = \sum_i s_i * h_i, where \tag{5.11}$$

$$h_i = \begin{cases} 1/l_i & \text{if } P_{goodFetch}(i) > \text{T} \\ ap_i(1 - ff(i)) & \text{otherwise} \end{cases}$$

To see the above, consider a time period $T$ over which accesses appear at a rate of $a$ arrivals per unit time. The number of accesses in time T has a Poisson distribution with mean $aT$. Let the relative access probability of an object $i$ is $p_i$. From the discussion in Section 5.4.1, we can infer that the number of accesses to an object $i$ in time period $T$ also follows a Poisson distribution with mean $ap_iT$.

The estimated bandwidth is

$$BW_{ss_d} = \frac{1}{T}\sum_i \sum_{k=0}^{\infty} P(k \text{ accesses to } i \text{ in } T \text{ time}) * \sum_{j=0}^{k} P(j \text{ hits in those } k \text{ accesses})(k-j)s_i \tag{5.12}$$

$$P(k \text{ accesses to } i \text{ in } T \text{ time}) = e^{-\lambda}\frac{\lambda^k}{k!}, \text{ where } \lambda = ap_iT \quad (5.13)$$

$$P(j \text{ hits in } k \text{ accesses to } i) = C_j^k(p_i')^j(1-p_i')^{k-j}, \text{ where} \quad (5.14)$$

$$p_i' = \text{probability of hit on an access to } i$$

The probability of hit on an access to object $i$ is same the freshness factor calculated in Section 5.4.1. From above equations,

$$\begin{aligned} BW_{ss_d} &= \frac{1}{T}\sum_i\sum_{k=0}^{\infty} e^{-\lambda}\frac{\lambda^k}{k!}\sum_{j=0}^{k} C_j^k(p_i')^j(1-p_i')^{k-j}(k-j)s_i \\ &= \frac{1}{T}\sum_i\sum_{k=0}^{\infty} e^{-\lambda}\frac{\lambda^k}{k!}(k(1-p_i')s_i) \\ &= \frac{1}{T}\sum_i(1-p_i')(s_i)\sum_{k=0}^{\infty} ke^{-\lambda}\frac{\lambda^k}{k!} \\ &= \sum_i s_i(1-p_i')ap_i. \end{aligned} \quad (5.15)$$

### 5.4.2 Analytic model parameters

The synthetic workload simulator assumes a set of 1 billion objects that exhibit a Zipf like popularity distribution with the Zipf parameter $\alpha = -0.982$. The sizes of the objects are assumed to follow a *log-normal + pareto* like distribution as explained in [16]. We assume that there is no correlation between object sizes and their popularities. The distribution of object lifetimes was taken from [62] again assuming no correlation with popularity or size. The analysis in [28] support these assumptions of the non existence of any observable correlation between the popularity, lifetime and size of objects. Figures 5.6 and 5.7 shows the cumulative distribution functions for object sizes and lifetimes.

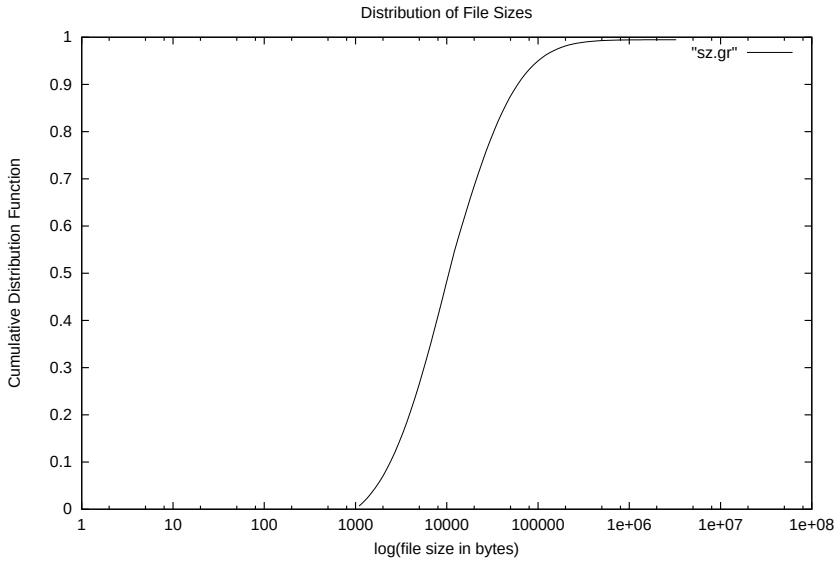



Figure 5.6: Cumulative distribution of object sizes

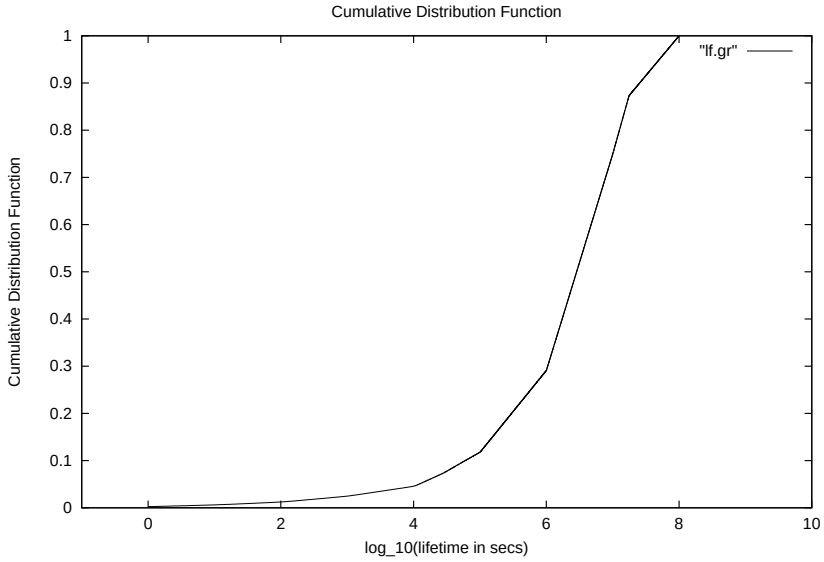



Figure 5.7: Cumulative distribution of object lifetimes

All data is considered to be cacheable. This assumption is justified since we are concerned only with the improvement in hit rate because of prefetching and uncacheable data affects both demand and long term prefetch caching alike. We also speculate that efforts such as active caches [33], active names [171], and ICAP [170] to support execution of server program at caches and Content Distribution Nodes. Also the efforts to improve cache consistency semantics [185, 49, 113, 123] will enable caching of much currently uncacheable data. To obtain the steady state hit rate we need to simulate a trace consisting of several billion records. Since this takes an inordinate amount of time, we resort to the expression derived in the previous section to compute the steady state hit rate of an infinite-size demand cache. The bandwidth consumption for the prefetching strategy is computed by summing $size_i/lifetime_i$ over all prefetched objects with ($P_{goodFetch} > Goodfetch$ ).

The above simulation methodology has several limitations. It ignores burstiness of the request traffic and approximates it by a fixed average arrival rate. This will directly affect the number of consistency misses as seen by a demand cache. Burstiness of request traffic can also hurt prefetching at a proxy since there may not be any bandwidth available for prefetching at the peak points of demand traffic. (On the other hand prefetching can also help burstiness by smoothing out demand.) Ignoring temporal locality in the synthetically generated trace underestimates the hit rate seen by the demand cache.

The above methodology models a scenario where the universe of objects being accessed at the cache is fixed and their popularities known *a priori*. However, the Internet is a dynamic set of objects with new objects being created continuously. But, our model and the prefetching strategy extends to a dynamic universe of objects by assuming the existence of an oracle to perform the statistics gathering as described in Section 5.2.1. Such an oracle continuously maintains the information about changing set of objects, their popularities and lifetime distributions. The

prefetching strategy obtains a snapshot of the Internet from the oracle and decides whether or not to prefetch an object solely on the basis of its current popularity and mean lifetime, independent of other objects and independent of its history. Thus, given good statistical data, it makes no difference if the collection of objects in the prefetch set change over time. As noted in Section 5.2.1, developing such a realistic statistics predictor module is a subject left as future work. Though such an oracle is unrealistic, we show in the next subsection that the prefetching strategy yields to an adaptive/learning implementation that shows attractive performance in practice.

### 5.4.3 Proxy trace simulation

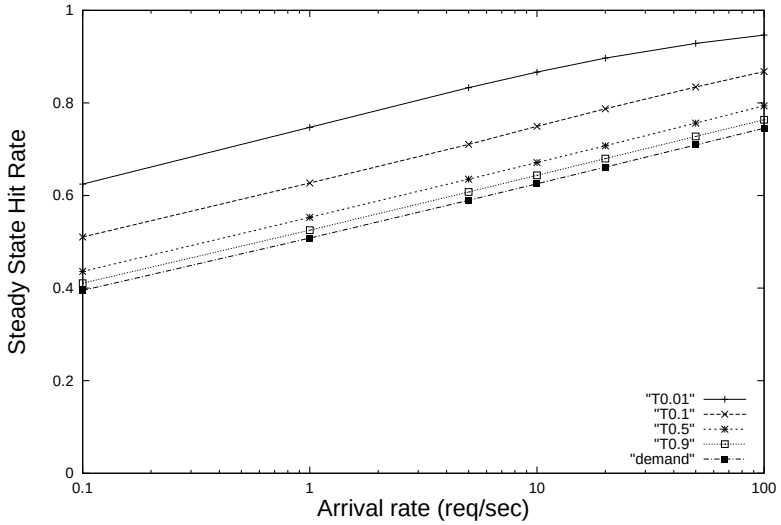
The trace based simulator uses a 12 day trace logged by the Squid proxy cache [31] at NCSA, Urbana Champaign between Feb 27 and Mar 10 2001. The trace consists of about 10 million records and accesses to 4.2 million unique objects. We simulate an LRU based demand cache and a prefetch cache implementing the Goodfetch algorithm using two simple predictors for assessing object popularities. Query URLs (with a "?" in them) are considered as both uncacheable and unprefetchable.

The sizes of the objects are used as logged in the trace. However, as in the analytic model, lifetimes are generated synthetically from the distribution given in [62], because the traces do not contain object update information. Since our prefetch strategy is sensitive to object lifetimes, this distribution could directly and significantly affect our results. Hence we also perform a sensitivity analysis of the performance of the Goodfetch algorithm with respect to median object lifetime by shifting the probability distribution curve of the lifetimes by several orders of magnitude along the lifetime axis.

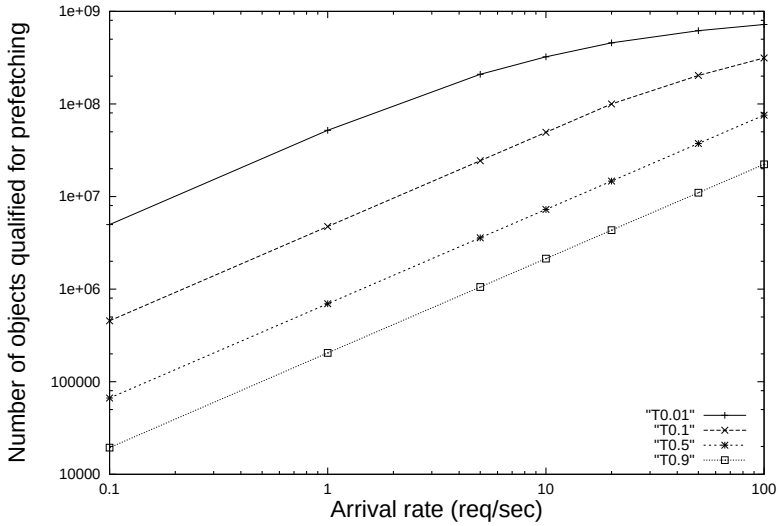
We use a simple statistical model to estimate the popularities of the objects. The simulator maintains a running count of the number of accesses to each object and computes object popularity by dividing this number by the total number of

accesses seen. At any access the simulator computes the $P_{goodFetch}$ of the object as defined in Equation 5.2 and checks if it exceeds the threshold, in which case the access is considered to be a prefetch hit. We call this object popularity predictor as *Predictor1*. *Predictor1* is aggressive, as it uses the current access to update the corresponding object's popularity before computing its $P_{goodFetch}$. This predictor can potentially save compulsory misses to an object since it knows an object's popularity even before the first access to it. Such a predictor is feasible in a scenario where popularity information is aggressively pushed out by the server, or is widely shared across several cooperating caches. We also simulate a more conservative predictor known as *Predictor-2* that can not prevent compulsory misses. An object has to be seen at least once before before its popularity can be used to consider it for prefetching by this predictor. For comparison purposes, we also simulate an oracle *EverFresh* that gets rid of *all* consistency misses, *i.e.*, an access is a hit if the corresponding object has been seen before. The hit rate so obtained is the maximum attainable by a cache based on local prefetching alone. Even the oracle cannot prevent compulsory misses or achieve hits to uncacheable objects. For all of the experiments, the cache was allowed to warm up for 8 days. The long period of warm up will prevent inflated estimates of popularities corresponding to accesses in the beginning of the trace.

As part of future work we intend to measure the performance of more aggressive predictors that gather popularity information from several cooperating caches. In such a scenario, it might be possible to considerably reduce compulsory as well as consistency misses.

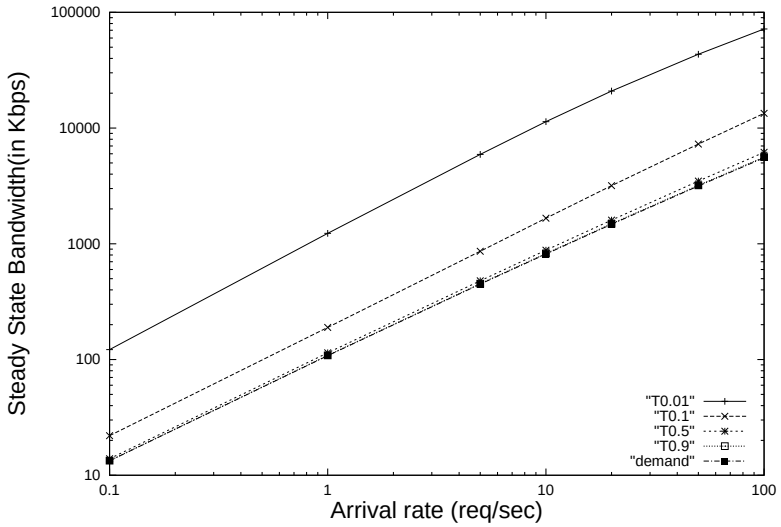



(a) Steady state hit rate

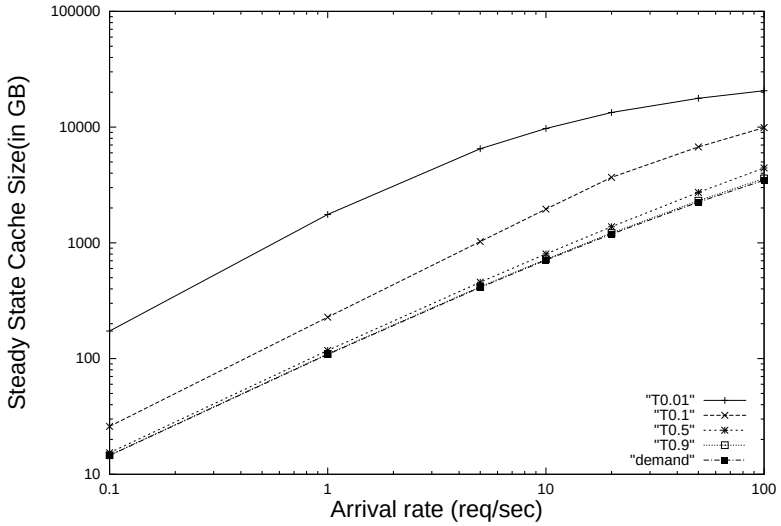



(b) Number of objects prefetched

Figure 5.8: Prefetching popular long lived objects. Effect of increasing request arrival rates for various thresholds




(a) Steady state bandwidth consumed




(b) Steady state cache size

Figure 5.9: Prefetching popular long lived objects. Effect of increasing request arrival rates for various thresholds

## 5.5 Results

### 5.5.1 Analytic Evaluation

Figure 5.8(a) plots the hit rates obtained by our prefetching policy for various prefetch thresholds with increasing request arrival rate. Note that demand request arrival rate is a key parameter describing the scale of the content distribution node under consideration and an increase in demand arrival rate increases the set of objects that meet a given threshold. The steady state hit rates for demand cache serve as a baseline for comparing the hit rates obtained by our prefetch policy. The graphs show that the threshold policy improves hit rates. For example, at an arrival rate of 1 req/sec, for threshold 0.5, we get an overall hit rate of 55% compared to a hit rate of 50% obtained by an infinite demand cache in steady state. For an arrival rate of 10 req/sec, for threshold 0.1, we get an overall hit rate of 75% compared to a hit rate of 62% obtained by the demand cache.

Note that significant improvements are achievable across a broad range of CDN scales. Although lower arrival rates reduce the collection of objects that meet the prefetch threshold criteria, lower arrival rates also reduce the steady state hit rates achieved by a demand cache.

Figure 5.8(b) plots the total number of objects (from our simulated universe of one billion objects) that qualify to be prefetched at various threshold values with increasing arrival rates. Figures 5.9(a) and 5.9(b) plot the amount of prefetch bandwidth and prefetch cache needed to maintain the prefetched objects.

As seen from the graphs, the threshold value affects the observed hit rate and the overhead. A high threshold implies a better chance of use of the prefetched object but a decreased hit rate since fewer objects will qualify for prefetching. A high threshold means that the amount of spare bandwidth in the system is less. Thus, the simulator limits bandwidth and cache size usage by prefetch requests through the threshold. As observed before,

Figure 5.9(a) suggests that as the arrival rate increases, even low thresholds would incur a modest bandwidth overhead as compared to demand bandwidth. For example, for an arrival rate of 10 req/sec, a threshold of 0.1 would incur bandwidth costs of 1.6Mbps over the demand bandwidth of 800Kbps. From Figure 5.8(a), this would give us a hit rate of 75%.

Figure 5.9(b) helps us in analyzing a typical cache size budget needed at a real world proxy given its request rate. For example, a 10req/sec request rate, with a threshold of 0.1 would correspond to a 2TB prefetch cache size. Given today's disk costs, it would cost around $6400 to add a 2TB disk. From Figure 5.8(a), a threshold of 0.1 at an arrival rate of 10 req/sec would provide us with a 75% hit rate.

From the graphs, at the cost of 10TB disk and 10Mbps bandwidth, we could achieve 87% hit rate which effectively means a decrease in missrate from 45% to 13%.

### 5.5.2 Trace-Based Simulations

A simple analyis of the trace showed that out of the 10.9 million requests that the proxy received, approximately 10% were consistency check messages. This implies that even an ideal prefetching strategy that prevents all of the consistency misses cannot give an improvement of more than 10% in hit rate over that of an infinite demand cache, unless, its statistics tracing spans multiple caches, or it allows servers to supply popularity estimates when objects are created.

In our experiments we assume a demand cache of size 28GB. We allow the cache to be warmed for 8 days and then gather performance measurements over the remaining 4 days.

Figures 5.10,5.11, and 5.12 show the change in hit rate, overall bandwidth consumption (demand+prefetch) and overall cache size (demand+prefetch) with

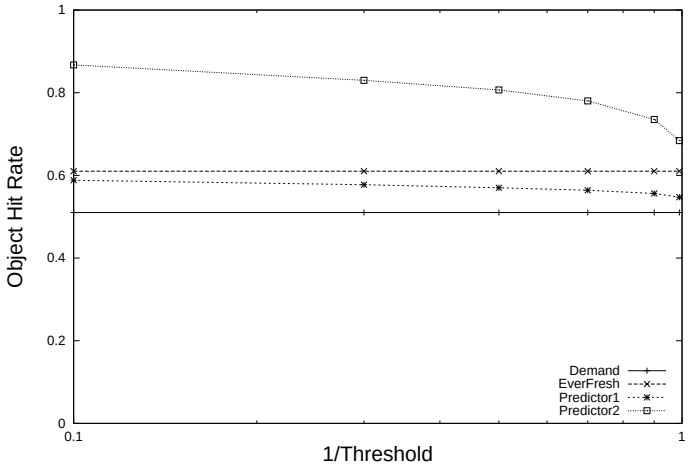



Figure 5.10: Object Hit rate vs. threshold

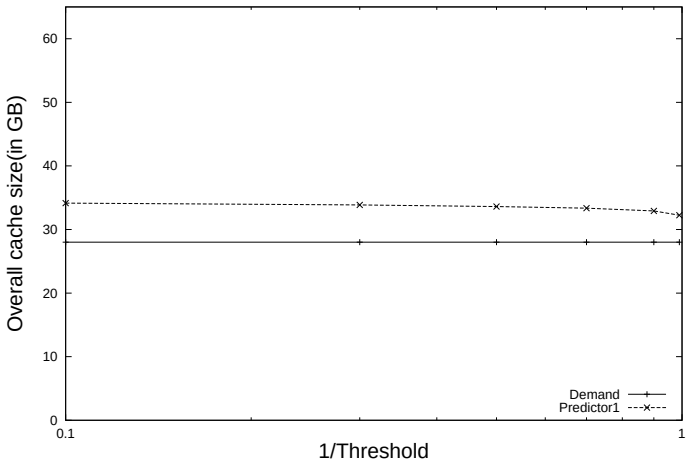



Figure 5.11: Overall cache size vs. threshold

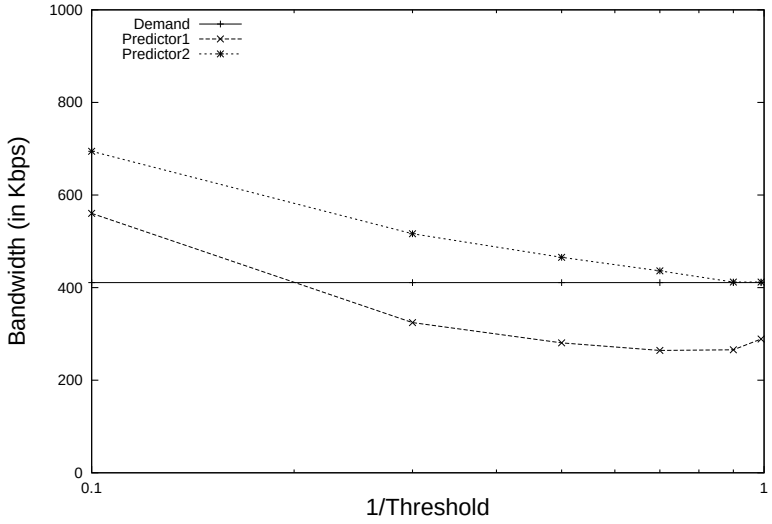



Figure 5.12: Total Bandwidth vs. threshold

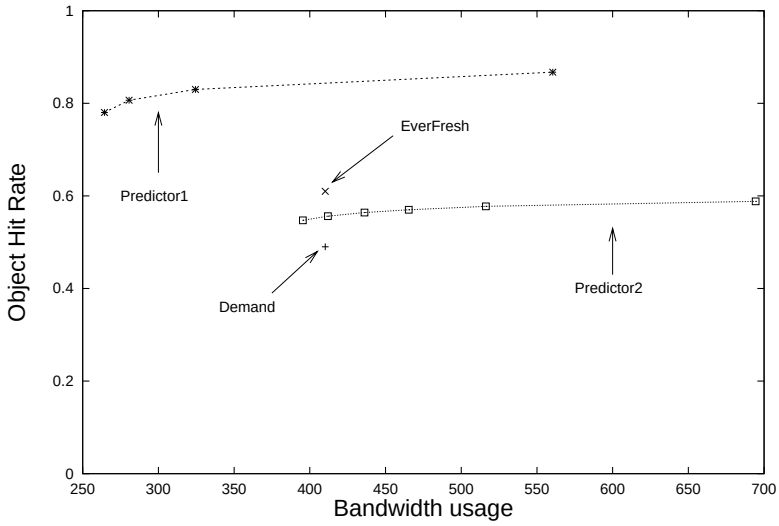



Figure 5.13: Hit rate vs. Total Bandwidth

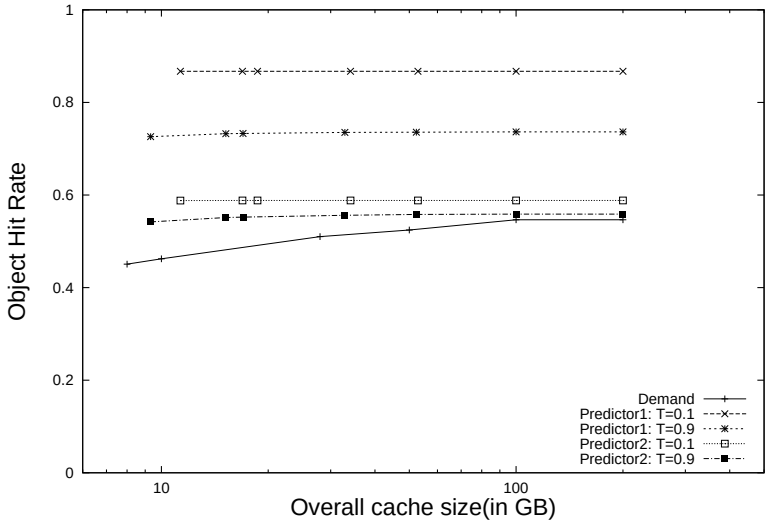



Figure 5.14: Hit rate vs. Total cache size

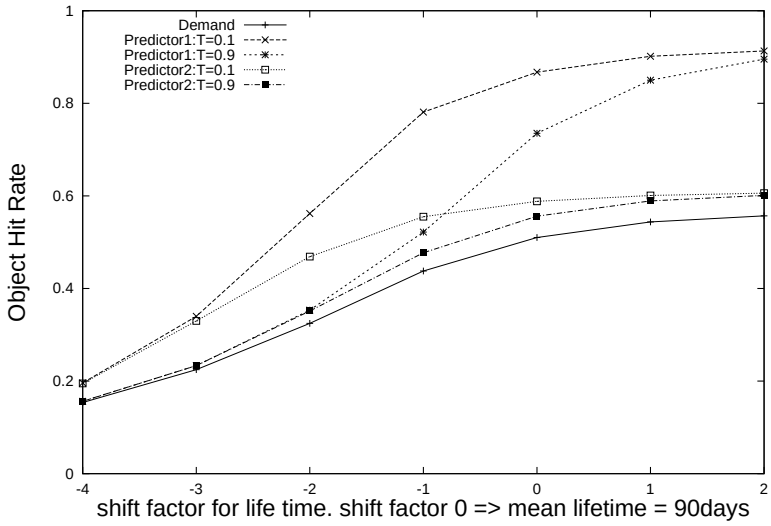



Figure 5.15: Object Hit rate senstivity

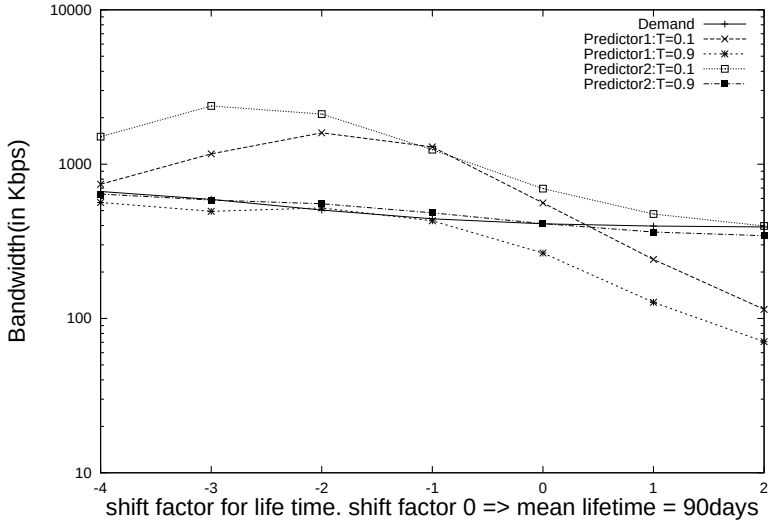



Figure 5.16: Bandwidth sensitivity

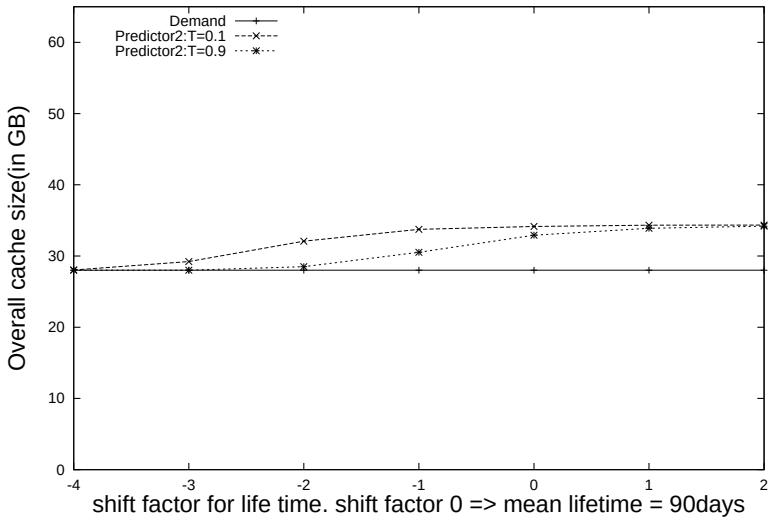



Figure 5.17: Total cache size sensitivity

varying thresholds. In figure 5.10, the *EverFresh* algorithm gives a hit rate of about 61%. At a threshold of 0.1 *Predictor2* almost matches this "optimal" achievable hit rate. *Predictor1*, as expected, gives much higher hit rates since it can avoid compulsary misses as well. Figure 5.12 shows that for a threshold of 0.1, the bandwidth blow-up as compared to the demand bandwidth is less than 2X. *Predictor 1* shows lesser bandwidth costs because of the *smoothing* out of prefetched bytes over a larger duration (a typical prefetched object's lifetime) as opposed to the post-warming period of 4 days for *Demand* and *Predictor2*. Figure 5.11 shows that the increase in cache size due to prefetching is nominal as well. Thus, we conclude that we can obtain significant improvements in hit rate at modest bandwidth and cache space costs.

Figures 5.13 and 5.14 show the attainable hit rate with respect to the bandwidth and cache space costs respectively. These graphs have also been generated from the data obtained from experiments performed by varying the threshold and have been included for ease of reference.

To test the sensitivity of our results to our assumptions about lifetimes, we vary the mean life time of objects and study its effect. Figures 5.15, 5.16, 5.17 show our results. The x-axis shows a shift factor $s$ which denotes the horizontal displacement along the lifetime axis (on a logscale) of the probability density function corresponding to the CDF taken from [62]. This graph varies the mean lifetime of the objects across several orders of magnitude, with each unit representing a change in the average lifetimes by a factor of 10. The graphs show that when life times are small, we get less hit rate improvement but at the same time use less prefetch cache size and bandwidth. This result is expected because our algorithm does not select short lived objects. At bigger lifetime values, we achieve higher hit rates at a reduced cost of bandwidth and prefetch cache size. The hit rate graphs at high threshold values flatten as the number of unique objects are limited, and almost all the objects

would have already qualified for prefetch. This observation holds for a universe of a fixed set of objects; if we are already caching all the ojects, then increase in life times or arrival rates would not alter hit rates. But the bandwidth required to keep the objects refreshed reduces proportionally as the lifetimes increase.

In summary, trace based simulation results show that our prefetch algorithm indeed provides significant hit rate improvements. One limitation of our trace based study is that we chose a medium-sized trace. But given the results we obtained, increasing the trace length would only benefit our results rather than hurt them.

## 5.6 Related Work

The idea of prefetching in the web has been widely studied by many researchers recently. Most of the research has concentrated on short-term prefetching based on recent access patterns of clients. Duchamp [64] provides a survey of various research contributions relevant to short-term prefetching in the web: software systems, algorithms, simulations and prototypes [140], and papers that establish bounds [114]. Duchamp [64] proposes to prefetch hyperlinks of web pages based on aggregate access patterns of clients as observed by the server. This approach gives priority to popular URLs and hence is similar to our popularity algorithm. Our (threshold based) approach is different from the above approaches in that we consider long-term prefetching suitable to CDNs [4] and busy proxies by prefetching objects that are both popular and long lived so that we get long-term benefits.

Gwertzman et al. [88] discuss push-caching which makes use of a server's global knowledge of usage patterns and network topology to distribute data to co-operating servers. A number of research efforts support multicast delivery for web content distribution. Li et al. [121] investigate multicast invalidation and delivery of popular, frequently updated objects to web cache proxies. Nayate et al. [135] discuss push-caching for a data dissemination service where updates are made by exactly

one server while replicas serve sequentially consistent data. These techniques are complementary to our work and can be used to realize the distribution of updates that we assume in our model.

Implementations of cooperating caches are becoming increasingly commonplace. Sharing of popularity information across caches enables not only better estimates of popularity but can also prevent some compulsory misses. In the next chapter, we show how to select objects to be prefetched in a cooperative caching environment to minimize average access cost.

## 5.7 Discussion

In this paper, we focussed on a technique called long-term prefetching, which is beneficial for web proxies and content distribution networks. In this prefetching model, we have explored an algorithm for object selection based on the popularity and lifetime of objects. We evaluated the performance of a long-term prefetching algorithm whose aggressiveness can be tuned by varying a prefetch threshold. For example, our analytical evaluation showed that, for a cache that receives 10 demand requests per second, our prefetching algorithm can improve hit rates by up to 13 percent, while increasing the bandwidth requirements by just over a factor of 2. Our trace-based results show that using simple history based predictors one can eliminate a significant fraction of consistency misses. More sophisticated popularity predictors that share information between cooperating caches can result in up to a 30% improvement in hit rate at a bandwidth blow-up of less than a factor of two.

Goodfetch or the probability of access before update is a global metric of an object's prefetch worthiness at a replica and can be compared across objects residing at different servers or copies at peer caches. Long-term prefetching can be naturally integrated with a self-tuning speculative replication system like NPS; speculatively replicated objects can be prioritzed using their goodfetch values at the server and

in the network. Long-term prefetching is ideal for proxies or content distribution servers as Goodfetch can be easily computed using aggregated request statistics, however, it is applicable at any replica that has means of estimating Goodfetch values for prefetched objects.

Short-term prefetching can be viewed as a special case of long-term prefetching where additional information is available to compute goodfetch values of objects. In Section 5.3, the expression $1-(1-p_i)^{a.l_i}$ for goodfetch was derived by modeling request sequences as independent arrivals. Often, servers can easily compute probabilities of access of other documents it serves given the current document the client is viewing. These probabilities could be significantly higher that the long-term average popularities of these objects observed at a proxy or the client, and hence, such objects can be attractive candidates for short-term prefetching. Nevertheless, the prefetch worthiness of such short-term prefetches is also determined by the probability of access before update. In particular, if a server knows that a document $i$ is likely to be accessed with probability $q$ in the next $t$ seconds by a particular client, it can inform the associated proxy which can modify that object's goodfetch as $1-(1-q)(1-p_i)^{ar.(l_i-t)}$ (assuming no other request dependencies). Similarly, more complex request dependency information could be incorporated while computing goodfetch. One advantage of long-term prefetching over just performing short-term prefetching is that the former has greater freedom to spread prefetched bytes over a longer period of time than just the time before the next access by a user; such freedom is valuable for performing speculative replication in the background.

The question of how to share information between the server and a proxy or a server and a client has not been sufficiently addressed in this dissertation. If a proxy alone maintains statistics of user behavior, it cannot avoid compulsary misses to objects it hasn't seen before or stale objects whose popularity has suddenly increased. On the other hand, if a server is prescient of such popularity fluctuations

at a proxy, it can proactively push hints of future knowledge to the proxy, which in turn can prefetch those documents. On first glance these savings might appear insignifcant, as they save only the first miss in an object's lifetime. However, one must keep in mind that saving the first miss is the only benefit prefetching provides over just demand based caching. As has been shown above, such savings can result in significant reductions in miss costs. The downside of obtaining popularity information from a server is that the server itself can have a skewed perception of object popularities because of intermediate caches absorbing all but the first request in an object's lifetime. This effect, termed as the *trickle down* effect in [63] and recognized earlier in the context of multi-level caches in file systems [133], can result in the server providing misleading information to a proxy. A proxy must combine the server's estimate of the prefetch worthiness of the object with its own by taking into account the fraction of requests it absorbed, and therefore hid from the server. With multiple proxies, where some have accessed the object at least once and others haven't, combining popularity and request dependency estimates computed by the server and a proxy appears non-trivial. The question of easy deployability of mechanisms that exchange such popularity information is also important, though it appears that techniques similar to the ones used in NPS or alternatives suggested in the paper could be effectively employed. These issues of providing mechanisms to aggregate popularity information in cooperative environments is an avenue of future work.

# Chapter 6

# Bandwidth-Constrained Speculative Replication for Cooperative Caches

## 6.1 Introduction

The previous chapter addressed the problem of speculative replication in a bandwidth-constrained environment for a large stand-alone cache. In this chapter, we extend the study to the problem of bandwidth-constrained speculative replication to a cooperative environment. The goal is to speculatively place copies of objects at a collection of distributed caches to minimize expected access times from distributed clients to those objects; each cache is subject to a maximum bandwidth constraint and the probability of access of each object at the cache (or its Goodfetch value as introduced in the previous chapter) is given a priori. We develop a simple algorithm to generate a bandwidth-constrained placement by hierarchically refining an initial per-cache greedy placement. We prove that this hierarchical algorithm generates a placement whose expected access time is within a constant factor of the optimal

placement's expected access time. We then proceed to extend this algorithm to compute close to optimal placement strategies for dynamic environments.

## 6.2 Motivation

The goal of bandwidth-constrained speculative replication in a cooperative environment is to place copies of objects at a collection of distributed caches to minimize expected access times from distributed clients to those objects. In a cooperative caching model [58], a cache miss at one location may be satisfied by another cache in the system. Unlike a stand-alone cache, in a cooperative environment, one must decide not only what objects to speculatively replicate, but also where to place speculative copies. Thus, the bandwidth-constrained speculative replication problem is essentially a placement problem and will also be referred to as the bandwidth-constrained placement problem in the rest of the chapter.

A number of scalable request-forwarding directory schemes have been developed to enable large-scale cooperative caching in WANs [5, 69, 169] and commercial *content distribution networks* of cooperating caches have been deployed by companies such such as Akamai and Digital Island. Although traditional caches are filled when client *demand requests* miss locally and cause data to be fetched from a remote site, hit rates might be significantly improved by pushing objects to caches before clients request them [89, 174]. The *distributed cache placement problem* attempts to select which objects should be pushed to which caches in order to optimize performance. A number of researchers have examined the space-constrained placement algorithms – in which cache storage space places limits on what can be cached where – in local area networks [58, 71, 120] and wide area networks [126, 111]. However, little attention has been paid to bandwidth constrained placement in cooperative environments.

Unfortunately, for WAN replication, bandwidth constraints may be more

restrictive to replication than space constraints. A number of web-trace simulations have indicated that only modest cache sizes are needed to achieve maximum hit rates [65, 84, 169]. Furthermore, maintaining a pushed copy of an object in a cache consumes not only disk space but also network bandwidth to update that copy when the origin data changes. Gray and Shenoy [82] compare the dollar cost of transmitting an object across the Internet to the dollar cost of storing the object on disk and indicate that network costs will be greater than disk storage cost for objects whose lifetime is less than 13 months.

In this chapter, we extend Korupolu et al.'s [126] space-constrained placement algorithm to address bandwidth constraints. The bandwidth-constrained problem differs from the simple space constrained model used by Korupolu et al. in several important ways:

- The bandwidth-constrained cost model depends on object update frequency and must account for the on-demand replication of objects that naturally occurs as demand-reads are processed. Conversely, the space-constrained cost model does not depend on object-update frequency, and a good space-constrained placement may not be improved by on-demand replication because any new replica must displace a previously placed object.

- In a bandwidth-constrained placement model, replicas of data initially present in caches should be left there, while such initial copies have no advantage over other objects under the space-constrained model.

- After a good bandwidth-constrained placement is calculated, it may take considerable time to send objects to their caches. In fact, in some cases bandwidth-constrained placement may be a continuous process as copies of objects are pushed to caches at some low background bandwidth, new objects are created, and existing objects are updated. Thus, it is important for

a bandwidth-constrained placement algorithm to work well in a dynamically changing environment and provide good performance at intermediate points in its execution. Conversely, Korupolu et al.'s algorithm only seeks to optimize the cost of the final, complete placement and runs in a batch mode as opposed to an incremental one.

We extend Korupolu et al.'s algorithm and show that our *Fixed-$t_{fill}$* algorithm generates a final placement that is within a constant factor of the cost of a bandwidth-constrained optimal placement under empty-cache initial conditions. We then show that our *InitFill* algorithm also provides a final placement that is within a constant factor of the optimal even when considering an initial placement. Finally, we show that our *DoublingEpoch* algorithm generates series of placements that is continuously within a constant factor of the optimal placement at any time $t$, but that to do so, it expands the required bandwidth by at most a factor of 4. This means that the *DoublingEpoch* algorithm is well-suited to a dynamic environment where objects are updated, demand-read copies appear, and new objects appear because it requires no *a priori* estimate of what point in time for which to optimize the placement schedule.

Our placement algorithms have several limitations. First, the constant cost and bandwidth-expansion factors that bound our worst case performance appear large (about a factor of 14 for the cost bound and 4 for the bandwidth expansion bound). However, previous experimental evaluation of the space-constrained placement algorithm indicates that for practical topologies and workloads, its behavior closely approximates the ideal algorithm despite similarly large constants [111]; our intuition leads us to expect similar behavior in the bandwidth constrained case. Second, although the *DoublingEpoch* algorithm is robust in a dynamic environment and provides a constant-factor approximation of the optimal bandwidth-expanded placement *within* any interval across which it is run, it must be restarted when

system conditions significantly change, and it is not provably near-optimal *across* a series of executions. Finally, all of our constant-factor bounds assume uniform unit-sized objects. The nonuniform-size placement problem is NP-hard. We present a simple heuristic that extends our algorithms to accommodate variable-sized objects. Future work is needed to experimentally validate our conjectures that for practical topologies and workloads our *Fixed-$t_{fill}$*, *InitFill*, and *DoublingEpoch* algorithms will be nearly optimal within any interval, that our heuristics for chaining the *DoublingEpoch* algorithm across intervals work well, and that our heuristics for variable-sized objects work well.

The rest of this chapter is organized as follows: In section 6.4, we describe the system architecture and the hierarchical distance and cost model we are considering. In section 6.5, we introduce the basic *Fixed-$t_{fill}$* algorithm that is a straightforward extension of the Greedy algorithm in [126] and prove that its cost is within a constant factor of the optimal. In section 6.6, we remove the restrictions assumed while developing the *Fixed-$t_{fill}$* algorithm and extend it to relate to more realistic scenarios. Herein, we introduce the Doubling Epoch algorithm and and analyze it for performance bounds. In section 6.7, we explain a heuristic algorithm to handle variable sized objects. We show that that the placement problem for variable sized objects reduces to the partition problem and is hence intractable and unyielding to efficient approximation algorithms.

## 6.3 Related Work

The placement problem has traditionally been treated as one constrained by space. Korupolu *et al.* [126] studied the problem of coordinated placement for hierarchical caches with space constraints, *i.e.* fixed cache sizes. They proved that under the hierarchical model of distances, the space constrained *Amortized Placement* algorithm is always within a constant factor (about 13.93) of the optimal. Though for practical

purposes this factor is rather large, the experimental work in [111] suggests that this algorithm yields an excellent approximation of the optimal for a wide range of workloads. In addition, the simplified greedy version of the amortized algorithm introduced in [126] has also been shown to provide an excellent approximation of the optimal, though in theory its performance can be arbitrarily far from optimal.

In an earlier study, Awerbuch, Bartal and Fiat [13] provide a polylog($n$)-competitive on-line algorithm for the general space constrained placement problem under the assumption that the size of each cache in the on-line algorithm is polylog($n$) times larger than the size in the optimal algorithm. The placement problem for a network of workstations modeled as a single level hierarchy has been studied by Leff, Wolf and Yu [120]. They provide heuristics for a distributed implementation of their solution. However, the heuristics make use of particular properties of single-level hierarchies and are not applicable to arbitrary hierarchies.

Replacement algorithms attempt to solve the problem of determining which object(s) are to be evicted when a cache miss occurs. Relevant studies of replacement algorithms have been done in [32, 186]. In the space constrained scenario, replacement algorithms may also be viewed as placement algorithms starting with an empty placement. The work in [111] shows that the hierarchical version of the *GreedyDual* replacement algorithm exhibits good cooperation and performs well in practice. However, for the bandwidth constrained placement problem, objects can be allowed to remain in the caches until they are modified, and hence replacement policy is not an issue.

The object placement problem has an orthogonal counterpart, namely the *object location* problem. The object location problem has been widely studied [25, 40, 164]. Recent studies such as Summary Cache [69], Cache Digest [155], Hint Cache [169], CRISP [148] and Adaptive Web Caching [190] generalize from hierarchies to more powerful cache-to-cache cooperation scenarios. Some recent

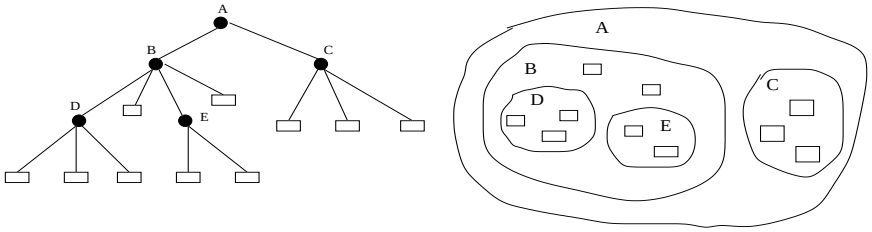



Figure 6.1: Model for network distances

studies have combined the location problem with the placement problem by remembering routing information at intermediate nodes [90, 127, 148, 182]. However, in this paper, we do not deal with lower level routing issues and separately consider only the placement problem.

To the best of our knowledge, the placement problem with bandwidth constraints has not been studied for hierarchical caching networks.

## 6.4 System Architecture

The system architecture is modeled as a set of $N$ distributed machines and a set $S$ of origin servers connected by a network. Assume that these machines are accessing a set of $M$ shared objects maintained at any of the servers in $S$ and cached at the machines. For each machine $i$, there is a fixed available bandwidth denoted by $bw(i)$ to push objects into the cache at machine $i$. The size of the cache at every machine is assumed to be very large. The cost of communication between any pair of machines $i$ and $j$ is given by the function $dist(i, j)$.

Requests for objects are made by clients at or near these machines. If there exists a local copy of a requested object $\alpha$ at machine $i$, then the object is served locally. If not, a directory (*e.g.* a summary [69] or a hint cache [169]) is consulted to find the nearest copy of object $\alpha$, which is fetched and served to the requester as well as stored locally at machine $i$. This implies that all future requests for object $\alpha$

at machine $i$ will be local hits. Thus the cost of satisfying an access request for an object $\alpha$ at a machine $i$, denoted by $c(i, \alpha)$ is given by the cost of communication $dist(i, j)$ between $i$ and $j$, where $j$ is the closest machine that possesses a copy of object $\alpha$. If no copy of object $\alpha$ resides on any of the caches in the network, then $c(i, \alpha)$ is defined to take a value $\Delta$ that denotes the *miss penalty*. In other words this is the cost of obtaining a requested object directly from an origin server rather than from one of the $N$ cooperating caches. Note that $\Delta$ must at least be as large as the maximum value of the function $dist$.

An example of a system with such properties is a large scale content distribution network where bandwidth costs dominate storage costs and where the caches have a limited available bandwidth to receive object updates.

### Hierarchical distance model

To make this problem tractable and applicable for practical distributed networks, we structure the distance function $dist(i, j)$ according to a hierarchical model. This hierarchy may be understood as a multilevel cluster or a *cluster tree* as shown in Fig 1. Such an organization of machines naturally reflects the way wide area network topologies are structured. For example in a collection of universities each machine belongs to the department cluster, which in turn belongs to the university cluster, and so on.

A *depth-0 hierarchy* is a singleton set containing one machine. A *depth-d hierarchy* $H$ is a virtual node $H$ plus a set of hierarchies $H_1, H_2, \ldots, H_k$, each of depth smaller than $d$, and at least one of which has depth exactly $d-1$. The hierarchies $H_1, \ldots, H_k$ are referred to as the children of $H$ and $H$ itself as the parent of $H_1, \ldots, H_k$. A hierarchy is associated with a diameter function $diam$ which is defined as follows: If $H$ is a singleton hierarchy, $diam(H) = 0$, else $diam(H) \geq \lambda diam(H_i)$, for all children $H_i$ of $H$. (Such a hierarchy is said to be $\lambda - separated$.)

For any hierarchy $H$, $machines(i)$ is defined to be the set of singleton hierarchies that are descendents of $H$. The distance function $dist(i,j)$ between any two machines $i$ and $j$ belonging to $H$ is defined as $diam(H_1)$ where $H_1$ is the least depth hierarchy such that $i$ and $j$ belong to $machines(H_1)$.

Thus, a hierarchy is essentially a tree and we use the two terms interchangeably. It must be remembered that while understanding a hierarchy as a cluster tree, the caches/machines are only at the leaf nodes. The intermediate nodes are virtual and only maintain book-keeping information. The distance function described above ignores small variations in cost inside a subtree and models a system where each machine has a set of nearby neighbors, all at approximately the same distance $d_1$, and then a set of next closest neighbors all at approximately a distance $d_2$ and so on. Note that the term *hierarchical caches* used to describe caching systems like Harvest [25], Squid [179] etc. is different and the two models must not be confused. In the latter, the hierarchy models the actual network topology and interior nodes act as caches. In our model the hierarchy itself is a logical tree introduced solely to capture network distances.

The model described above is the same as the ultrametric model used by Karger et al. [105] Such a hierarchical structure can be used to capture a wide variety of distributed networks like intranets or WANs. A local area network for example is a *depth-1* hierarchy. It can also be used to model content distribution networks - the leaf caches are the geographically distributed pseudo-servers. A client request originating at an ISP is directed to one of these caches that may either serve the object or redirect the request to the closest neighboring cache which stores a copy. If none of the caches store the object, then the request is directed to the origin server for the object. It seems reasonable to think that a hierarchal distance model may serve as a reasonable approximation of a complex network such as the Internet. In fact recent results on approximation of general metrics by tree metrics

[17, 20, 41] imply that any hierarchal placement algorithm may be used to obtain a placement algorithm for any arbitrary metric cost model with at most a blow up in the approximation factor by a polylog factor in the number of nodes.

### Cost Model

A *placement* assigns copies of objects to machines in a hierarchy. A *copy* $(i, \alpha)$ is a pair consisting of a machine $i$ in the hierarchy and an object $\alpha$. Every object $\alpha$ is associated with an expiry time denoted by $expiry(\alpha)$ that represents the point of time at which the object will be invalidated.[1]

For any machine $i$ and object $\alpha$, let $p(i, \alpha)$ denote the *probability* of access of object $\alpha$ at machine $i$ before $expiry(\alpha)$. For any hierarchy $H$, the *aggregate* probability of access of an object $\alpha$ $p(H, \alpha)$ is defined as $1 - \Pi_{i \in machines(H)}(1 - p(i, \alpha))$, *i.e.* it represents the probability of access of object $\alpha$ at at least one of the leaf nodes in the tree rooted at $H$. The goal of the coordinated placement strategy is to push objects into the caches such that the overall cost of access to these objects is minimized.

In order to develop a notion of the overall cost of a placement given the probabilities of access of objects at the machines, we observe that if there is a request for an object $\alpha$ in any hierarchy $H$, none of whose machines have a copy of $\alpha$, the incremental cost of leaving $H$ to fetch a copy of the $\alpha$ is $p(H, \alpha) \cdot (diam(parent(H)) - diam(H))$. The overall access cost $cost(P, H, \psi)$ of a placement $P$ over a hierarchy $H$ and a universe of objects $\psi$ is recursively defined to be $\sum_{\alpha \in \psi} p(h, \alpha) \cdot \delta(h, \alpha) \cdot (diam(parent(h)) - diam(h)) + \sum_{i \in [1,k]} cost(P, H_i, \psi)$, where $H_1, \ldots H_k$ are the immediate children of $H$ and $\delta(h, \alpha)$ takes the value 0 if at least one machine in $machines(h)$ contains a copy of $\alpha$, else 1. A formal justification of the cost function defined above may be arrived at by a straightforward proof of the

[1] In practice, $expiry(\alpha)$ may be known or estimated *a priori, e.g.* a newspaper website may be updated every night at 2am or past update patterns may be used to predict the next update.

claim that for a random series of $n$ object requests in accordance with the distribution of the given probabilities of accesses, the expected total latencies encountered by two given placements are in the ratio of their costs as defined above, as $n \to \infty$.

In practice, access probabilities may be known or estimated using application specific benchmarks with knowledge of history based statistical information [36]. Since objects are associated with lifetimes and probabilities of access are likely to show temporal variation, it seems natural to introduce a *fill-time* $t_{fill}$ within which the placement has to be accomplished. Thus, given probabilities of access and the available bandwidth to push objects into machines, the goal of the bandwidth constrained placement problem is to compute the placement with the minimum cost that can be accomplished within the fill-time.

## 6.5 Algorithms

In this section we present some core algorithms to handle the bandwidth constrained placement problem as defined above as well as its extensions to more realistic scenarios. We start with a basic algorithm called the *Fixed-$t_{fill}$* algorithm that solves the version of the placement problem defined in section 6.4 , *i.e.* it assumes that (i) all the caches are initially empty, (ii) objects do not get modified during $t_{fill}$, (iii) all object requests occur after the placement is complete, (iv) the probabilities of access and the universe of objects are fixed, and (v) all objects are of equal size. We first describe a simple greedy strategy that that captures the core idea of the *Fixed-$t_{fill}$* algorithm. We then describe an amortized version of this algorithm which adds a small technical adjustment needed for bounding the algorithm's worst case performance with respect to the optimal placement. Subsequently, in section 6.6 and 6.7 we proceed to progressively refine this algorithm to handle (i) non-empty initial placements, (ii) simultaneous placement and accesses to objects, (iii) object updates and introduction of new objects and (iv) variable sized objects.

All the algorithms described in this section have been described, for simplicity's sake, under the assumption of a centralized implementation. However, it is straightforward to transform the centralized implementation into an efficient distributed one by a procedure that parallels the approach outlined in [126].

### 6.5.1 The Greedy *Fixed-$t_{fill}$* Algorithm

The algorithm we present involves a bottom-up pass along the cluster tree. It starts with a tentative placement in which the caches at the leaves of the cluster tree pick the locally most valuable set of objects. Then the algorithm proceeds up the hierarchy by having each node improve the corresponding subtree's placement.

It must be emphasized that the problem of determining a good placement is conceptually distinct from accomplishing that placement itself. The latter involves routing issues which we do not address directly in this discussion. These issues are simply abstracted out in terms of an effective bandwidth $bw(i)$ available to push objects into machine $i$'s cache.

We first introduce a few definitions. For any $H$, and a placement $P$ over it, we define an object $\alpha$ to be *P-missing* if no copy of $\alpha$ exists in any of the caches in $H$. The benefit of an object $\alpha$ at a machine $i$ in a placement $P$ is defined as the increase in the cost of the placement over $H$ were $\alpha$ to be dropped from $i$, and is denoted by $benefit(i, \alpha, P)$. It follows from the definition that the benefit of a copy is dependent on where other copies of the same object are distributed among the machines in the hierarchy. The *value* of a *P-missing* object $\alpha$ in a hierarchy $H$ is defined to be $p(H, \alpha) \cdot (diam(parent(H)) - diam(H))$. The *Fixed-$t_{fill}$* greedy algorithm for bandwidth constrained placement is given below.

**Input:** A hierarchy $U$, the universe of objects $\psi$, a fill time $t_{fill}$, available bandwidth $bw(i)$ for $i \in machines(U)$, the access probability $p(i, \alpha)$ for all objects $\alpha$ and

$i \in machines(U)$. Let all objects be of size $objectsize$.

**Initialization:** For each $i \in machines(U)$, set $size(i) = bw(i) \cdot t_{fill}$. For each $i \in machines(U)$, select the $\lfloor \frac{size(i)}{objectsize} \rfloor$ objects with the highest local probability of access $p(i, \alpha)$ and call this local placement $P_i$. For each object $\alpha$ that gets selected at machine $i$, initialize $AssignedBenefit(i, \alpha) = p(i, \alpha) \cdot (diam(parent(i)) - diam(i))$

**Iterative step:** step $d$, $1 \leq d \leq depth(U)$

1. Compute the level-$d$ placement for each depth $d$ hierarchy $H$ as follows: Let $H_1, \ldots, H_k$ be the constituting hierarchies that are the immediate children of the hierarchy $H$. Initialize the placement $P_H$ over $C$ to the union of the placements already computed at $H_1, \ldots, H_k$, *i.e.* $P_H = \cup_{i \in children(H)} P_i$.

2. *Update benefits:* For each object $\alpha$ in $H$ that has one or more copies in the current placement $P_H$, let $(i, \alpha_p)$ be the *primary* copy, *i.e.* the copy with the highest local *AssignedBenefit*. The *AssignedBenefit* of this primary copy is increased by $H$'s aggregate access probability times the cost of leaving hierarchy $H$, *i.e.*, $AssignedBenefit(i, \alpha_p) \mathrel{+}= p(H, \alpha) \cdot (diam(parent(H)) - diam(H))$. All other copies of $\alpha$ are known as secondary copies and their $AssignedBenefits$ remain unchanged. Let $X$ denote the set of $P_H$-$missing$ objects.

3. *Greedy Swap Phase:* While there is a *P-missing* object $\beta$ in $X$, whose $value$ is more than the copy $(i, \alpha)$ with the least *AssignedBenefit* in $P$, remove the copy $(i, \alpha)$ and substitute a copy of $(i, \beta)$ with its *AssignedBenefit* initialized to its $value$ just before the swap.

**Object Insertion:** After the placement $P$ over the hierarchy $H$ has been computed, send out the selected copies of objects into the corresponding machines in the order of their $AssignedBenefits$ computed at the end of the loop above.

The key idea is that the swapping procedure at every level continues till there exists a *P-missing* object with value greater than the least *beneficial* copy in the current placement. The greedy swapping procedure stated above uses *AssignedBenefits* instead. However, observe that the *AssignedBenefit* represents the *benefit* of a copy just before it gets swapped out, and that the *benefit* of a copy is at most equal to its *AssignedBenefit*. It follows that the greedy swapping rule is equivalent to one based on swapping out the copy with the least *benefit*.

It may be verified that the execution time complexity of the greedy algorithm given above is $O(C \cdot N)$, where $C$ is the sum of the cache sizes at all of the leaf nodes and $N$ is the total number of nodes in the hierarchy.

### 6.5.2 The Amortized *Fixed-$t_{fill}$* Algorithm

Though the greedy algorithm seems simple and promising, the placement it computes in the worst case could be arbitrarily far from optimal. The fundamental drawback of the algorithm is that a single secondary copy of some object may prevent swapping in of several missing objects. Though the benefit of the secondary copy may be larger than the value of each of the missing objects, on the whole it might be much less than the sum of all these values put together. Korupolu *et al.* [126] demonstrate that for the space constrained version of the greedy algorithm, this effect can lead to a placement arbitrarily far from optimal.

In order to overcome this problem, we augment the greedy algorithm with an amortization step using a potential function identical to the one used in [126]. The potential function accumulates the values of all the missing objects, and the accumulated potential is then used to reduce the benefits of certain secondary items thereby accelerating their removal from the placement. The Amortized *Fixed-$t_{fill}$* algorithm is as follows:

**Initialization**: Same as greedy except that we also set a potential $\phi_i$ for each machine $i$ to zero.

**Iterative Step:** Same as in the greedy algorithm except that the potential $\phi_H$ is set to the sum of the potentials $\phi_1, \ldots, \phi_2$ computed by the children of $H$.

1. *Update Benefits:* Same as in the greedy algorithm.

2. *Greedy Swap Phase*: This procedure is similar to the swapping procedure in the greedy algorithm except that the potential $\phi$ is used to reduce the *AssignedBenefits* of some copies.

   1. Let $y_p$ be the primary copy with the least *AssignedBenefit* and $y_s$ the secondary copy with the least *AssignedBenefit* in $P_H$. Let $\alpha$ be the highest-valued object in $X$, the set of all $P_H$-$missing$ objects.

   2. If $value(\alpha) > min(AssignedBenefit(y_p), AssignedBenefit(y_s) - \phi_H)$, then perform one of the two following swap operations, and goto step 1.

      - If $AssignedBenefit(y_p) < AssignedBenefit(y_s) - \phi_H$, swap $y_p$ with $\alpha$. Set $X$ to $X - \alpha + \alpha'$, where $\alpha'$ is the object corresponding to the copy $y_p$. Set $value(\alpha')$ to $Assignedbenefit(y_p)$.

      - Otherwise, remove $y_s$ from $Q$ and substitute it with $\alpha$. Set $X$ to $X - \alpha$ and reset the potential to $max(0, \phi_H - Assignedbenefit(y_s))$.

   3. *Update Potential:* Add the values of all the $P_H$-$missing$ objects in $X$ to $\phi$.

**Theorem 1**: The cost of the amortized $Fixed - t_{fill}$ algorithm is within a constant factor of the cost of the optimal placement.

*Proof Outline:* The proof of optimality of the amortized bandwidth constrained placement algorithm exactly parallels the proof of its space constrained counterpart introduced in [126]. The fundamental difference between the cost model we developed and that used in [126] is that the *access quotient* in the former is based on probabilities of access while in the latter, it is the frequency of access. The cost of a placement in the former is defined as $\sum freq(\alpha, i) \cdot diam(closest(\alpha, i))$, over all $i \in machines(H)$ and $\alpha \in \psi$, where $freq(\alpha, i)$ denotes the frequency of access of object $\alpha$ at machine $i$ and $closest(\alpha, i)$ is the least common ancestor $v$ of $i$ such that at least one copy of the object $\alpha$ is present in $machines(v)$. We first claim that both the cost models are identical except for the definition of the aggregate access quotient at interior nodes.

**Lemma 1:** Let $f(\alpha, i)$ denote the frequency of access of object $\alpha$ at a leaf node $i$ in a hierarchy $H$. Let $f(\alpha, v)$ at an interior node $v$ denote the aggregate access quotient as defined in the frequencies model to be the sum of the frequencies of access of object $\alpha$ over all $i \in machines(v)$. Then,

$$\sum_{u \in H, \alpha \in \psi} f(u, \alpha) \cdot \delta(\alpha, u) \cdot (diam(parent(u)) - diam(u)),$$
$$= \sum_{i \in machines(H), \alpha \in \psi} f(u, \alpha) \cdot (dist(closest(\alpha, i))$$

The proof of the above lemma is by a straightforward rearrangement of terms. Thus, it follows that the amortized bandwidth constrained algorithm is within a constant factor of the optimal if the access quotient to an object at an interior node were defined as the sum of the access quotients to the same object over all leaf node descendents of that node.

The proof of optimality to within a constant factor in the frequencies case proceeds by introducing what is known as a bridging placement. The algorithm to compute the bridging placement is parameterized by a fixed, but arbitrarily chosen

placement, say $P$. The bridging placement, say $B$, is an intermediate placement that is then proved to be at least as expensive as the corresponding amortized placement and is also proved to be within a constant factor $\lambda.(1 + 3\lambda/(\lambda - 1))$ of $P$. Since $P$ could be arbitrarily chosen and hence chosen as the optimal placement, it follows that the amortized placement is within a constant factor of the optimal. All the properties of the aggregate access quotient as defined in the frequencies model also extend the probabilities model. In particular we state below a property that relates the benefit of a copy to its value just before it gets swapped in. This property is crucial in deriving the constant factor of approximation $\lambda(1 + 3 \cdot \frac{\lambda}{\lambda-1})$, and it is straightforward to show that it holds in the probabilities model as well.

**Lemma 2:** Let $H$ denote a $\lambda - separated$ hierarchy for some $\lambda > 1$, let $P$ denote an $H - placement$ in which a copy of an object $\alpha$ be placed at a machine $i$. Then $benefit(\alpha, i) \leq \lambda/(\lambda - 1).value(H, \alpha)$.

Though the worst-case constant (about 14) is large for practical purposes, measurements indicate that the space constrained variation performs nearly optimally for the workloads examined in [111], and we expect similar performance for bandwidth constrained workloads. In the rest of the chapter we just use the term *Fixed-$t_{fill}$* algorithm to mean amortized *Fixed-$t_{fill}$* algorithm.

## 6.6 The Dynamic Case

In the previous section we introduced a placement algorithm that is a straightforward extension of the space constrained amortized placement algorithm in [126]. However, we made several restrictive assumptions. In a content distribution network for example, we need to handle object updates dynamically. New copies of objects

have to be continuously pushed out to the caches to maintain consistency. The universe of objects itself may change with time and the objects may have arbitrary sizes. In this section we perform a stepwise refinement of the *Fixed-$t_{fill}$* algorithm that addresses most of these constraints.

We first present the *InitFill* algorithm that computes a placement given an existing set of objects already placed at the machines. To relate this to a practical scenario, imagine a content distribution network where we have placed a subset of objects at the caches or content servers over the course of an hour. At this point a significant number of objects could possibly change, along with their probabilities of access. For example, a sudden important news event at a news website could cause new articles with higher popularity to appear. Since not all of the objects might have changed, we would still like to leverage the benefit of the objects already placed in the network. However, the *Fixed-$t_{fill}$* algorithm introduced in section 6.5, assumes the caches to be initially empty. The space constrained placement algorithm on which we based the *Fixed-$t_{fill}$* algorithm does not take into consideration the objects already present in the cache. In general, the *InitFill* algorithm is useful for iterative application of the placement algorithm.

Next, we discuss the issues that arise when choosing $t_{fill}$ in a dynamic environment where objects could get modified during the placement, the universe of objects and associated access probabilities could change, requests for objects could occur simultaneously with the placement and the caches could start from a non-empty state. We show examples for which naive strategies for selection of the fill time, could lead to arbitrarily poor placements. We then present the *DoublingEpoch* algorithm that handles such dynamic object updates as well as modification of the access probability distribution and the universe of objects. We analyze this algorithm and prove that, given a 4X blow-up in bandwidth, this algorithm computes a placement whose cost is within a constant factor of the cost of the optimal.

The *DoublingEpoch* algorithm is therefore useful for continuous application of the placement algorithm in a dynamic situation.

### 6.6.1 Initial Placement

Assume that we already have an initial placement over a hierarchy $H$ given to us instead of empty caches. We modify the *Fixed-$t_{fill}$* amortized algorithm are as follows. We a) add sufficient free *virtual bandwidth* to each cache to store the already placed objects and b) artificially inflate the access probabilities of the already placed copies to force the algorithm to include them in the placement. The Following is the *InitFill* algorithm, given an initial placement $P_1$:

1. **Initialization:** For each machine $i$, set $size(i)$ to the sum of the combined size of the objects already present in the cache and $bw(i) \cdot t_{fill}$. For each copy $(i, \alpha) \in P_1$, set $p(i, \alpha) = 1$. Prioritize the copies in $P_1$ so that, ties while placing the first $\lfloor size(i)/objsize \rfloor$ objects at machine $i$ during the initialization phase are always broken in favour of copies in $P_1$. (Note that the greedy/amortized algorithm allows us to do this.) For each of the already placed copies $(i, \alpha)$, set $AssignedBenefit(i, \alpha)$ to $\infty$.
2. Run the rest of the *Fixed-$t_{fill}$* as before on this modified problem instance, resulting in a final placement $P$.
3. Delete from $P$, all copies $(i, \alpha) \in P_1$. (This is so that the already placed objects don't have to be pushed into their corresponding caches again during the *Object Insertion* phase.)

**Theorem 2**: The placement computed by the *InitFill* algorithm is within a constant factor of optimal.

*Proof*: It is clear that none of the initially placed objects ever get swapped out

during the course of the *Fixed-$t_{fill}$* algorithm, since their *Assignedbenefit* is set to $\infty$ at level 1. That they get placed at level 1 is ensured by setting their access probabilities to 1, prioritizing them and ensuring that sufficient bandwidth is available to place them. Let $P^{'}$ be the placement computed by the *InitFill* algorithm and $P^{'}_{opt}$ the optimal placement with the already placed objects with respect to the modified access probabilities. Let $P_{opt}$ denote the optimal placement with the already placed objects with respect to the unmodified access probabilities. Then, by the result of optimality to within a constant factor $c(\approx 14)$ proved in the previous section, $c \cdot cost(P^{'}) \leq cost(P^{'}_{opt})$. However, $cost(P^{'}_{opt}) = cost(P_{opt})$ in spite of the modified access probabilities for the already placed objects because the terms corresponding to the already placed copies anyway contribute to zero in either of the costs. It follows that $c \cdot cost(P^{'}) \leq cost(P_{opt})$. $\Box$

The *InitFill* algorithm is attractive since it allows for an incremental implementation of the static algorithm. We could start with the cache at each machine being empty at some initial time $t_0$ and then onwards at various points of time, invoke the *InitFill* algorithm to recompute a placement.

We also point out that the *InitFill* algorithm may be used to compute close to optimal placements over a distance model wherein the servers are located at different distances from different caches and have varying performance properties. To accommodate this extension, for each server create a new dummy cache located in the network topology at the same place as the server and include each of these dummy caches in the virtual cluster hierarchy in the same manner as regular caches. Set the bandwidth of each dummy cache to be 0 and initialize the contents of each dummy cache to include the objects served by the corresponding server. The *InitFill* algorithm may now be used so that the cost of accessing an object from a dummy cache matches the miss cost of fetching it from the server.

### 6.6.2 Challenges in choosing $t_{fill}$

The discussion above assumes that a time epoch $t_{fill}$ is known *a priori*. However, in real systems, it is potentially difficult to choose $t_{fill}$ optimally. First, in some systems it may be difficult to predict when the universe of objects being placed (or their probabilities of access) will change. Second, even for a static set of objects and access probabilities, the placement algorithms in the previous section optimize the performance of the placement achieved *after* $t_{fill}$ time has elapsed. If placement and client reads proceed in parallel (which may often be the case in real systems), then reads during the $t_{fill}$ interval may see substantially sub-optimal performance during this transient interval. In this section we investigate the problem of minimizing the transient latency cost during the course of the placement itself and motivate the need to select an optimal sequence of epoch times to achieve the same.

Choosing too long a $t_{fill}$ interval and then sending out the selected objects in the order of their *AssignedBenefits* to the individual caches may cause the transient access cost of the placement to be arbitrarily far from optimal. We illustrate this point using a specific example below. The transient access cost of a placement is defined as the average response time to service a request over the course of the complete interval.

*Claim 1*: Let $P_T(t), 0 \leq t \leq T$ be the placement strategy that computes a placement at time 0 using the *Fixed-$t_{fill}$* algorithm with a fill time $T$, and thereafter sends out the selected objects to the individual caches over the entire interval $T$ in the order of their *AssignedBenefits* so that at any time $t$, $\lfloor t \cdot bw(i)/objsize \rfloor$ objects have been placed at cache $i$. Throughout the epoch of length $T$, requests are served by the system according to the probability distribution specified. There exist topologies and access probability distributions for which the transient access cost of $P_T(t)$ could be arbitrarily far from the optimal achievable.

*Example*: Consider a topology consisting of a single level hierarchy with $n$ machines numbered 1 to $n$, diameter 1 and miss penalty $n$. Assume that there exist $n$ data objects $\alpha_1, \alpha_2, \ldots, \alpha_n$ and that the access probability distribution is as follows: $p(1, \alpha_i) = P,\ 1 \leq i \leq n$ and $p(j, \alpha_i) = 0,\ 2 \leq j \leq n, 1 \leq i \leq n$. We define a *time-step* as a small value of $t_{fill}$, say $\delta_{fill}$, such that for each of the leaf nodes $i$, $bw(i) \cdot \delta_{fill} \geq objsize$. Let the probability of access of object $j$ at machine 1 in any time-step be $\rho$ [2]. Further, assume that $T = n \cdot \delta_{fill}$

The $T$ epoch placement algorithm places one copy of all the $n$ objects on machine 1. The cost of this final placement is 0. However, this placement takes $n$ time-steps to complete and the cost of the transient placement[3] at step $i$ for object $j$ is given by 0, if $j < i$ (because object $j$ would have already been placed at the $j$'th time-step), and $(1-\rho)^i \cdot \rho \cdot n$ otherwise. The term $(1-\rho)^i \cdot \rho$ represents the probability that there is a request for object $j$ at machine 1 for the first time in step $i$. Therefore, the average response time to service a request for object $j$ is given by $\sum_{i \in [1,j]} n \cdot (1-\rho)^i \cdot \rho = n \cdot (1-(1-\rho)^{j+1})$. The transient access cost of $P_T(t)$ is thus:

$$AccessCost_1 = \frac{1}{n} \sum_{j \in [1,n]} n \cdot (1 - (1-\rho)^{j+1}) = O(n)$$

A better strategy is to place a copy of object $i$ at machine $i$ during step 1, and then to place a copy of object $i$ at machine 1 during step $i$. For this placement, the cost to service a request for object $j$ in step 1 is $\rho \cdot n$ and is bounded by $\rho \cdot 1$ in subsequent steps. The average response cost for a request for to object $j$ is at most $\frac{1}{n}(\rho \cdot n + n \cdot \rho \cdot 1) = \rho + 1$. Thus, it follows that the transient access cost of this placement strategy is

$$AccessCost_2 = \frac{1}{n} \sum_{j \in [1,n]} (\rho + 1) = O(1)$$

[2] In practice $\rho$ could be related to $P$ as $1 - (1-\rho)^\eta = P$, where $\eta$ is the number of time-steps in the object's lifetime

[3] a demand placement that happens to satisfy a request for an unavailable object.

Thus, from the above it is clear that the transient access cost of $P_T(t)$ to could be arbitrarily far from the optimal. $\Box$.

From the above example, it is clear that choosing $t_{fill}$ to be too long may lead the system to defer placing important objects because a *more valuable* location will later become available, resulting in a higher transient miss rate and therefore a higher average response time per request. This suggests that even if objects are static, *i.e.* they do not change, the naive strategy of computing a placement for a huge epoch and then pushing out objects in the order of their final *benefits* may not be efficient with respect to the time-averaged access cost of the placement.

Conversely, choosing $t_{fill}$ to be too short can cause the system to waste work by placing a copy at a sub-optimal location before placing it at the *right* place at a later time. The system therefore may end up taking more overall bandwidth or time (number of epochs) to complete a placement that is as good as the optimal placement for a larger epoch. We show this formally below again with an example:

*Claim 2*: Let $P$ be the placement computed by the *Fixed-$t_{fill}$* algorithm for a given fill time $T$. Let $\delta_f$ denote a tiny epoch such that for all machines $i$, $\delta_f \cdot bw(i) \geq objsize$. Then the number of iterations of the *InitFill* algorithm required to compute a placement that is at least as good as $P$ could be as many $T/\delta_f \cdot log(n)$, where $n$ is the total number of machines in the hierarchy.

*Example*: Consider a topology consisting of a single level hierarchy with $n$ machines numbered $1, \ldots, n$, diameter $d$ and miss penalty $\Delta$. Assume that the set of objects and the distribution of probabilities of access to objects are as follows: (i) every object is accessed at exactly one machine, there are $k$ distinct objects accessed at every machine. (ii) the probability of access of any object $\alpha$ at machine $i$, $p(i, \alpha) > \frac{\Delta}{d} \cdot p(i+1, \beta)$, where $\beta$ is any object accessed at machine $i+1$.

Assume further that all machines have the same bandwidth and every iteration of the $\delta_f$ epoch algorithm allows at most one object to be inserted into any machine. Assume $T/\delta_f$ to be $k$, so that the cost of the $T$ epoch placement is 0 (every object gets placed where it is accessed). At the end of the first iteration of the $\delta_f$ epoch InitFill algorithm, machine $i$ possesses a copy of the object with the $i'th$ highest probability of access. Thus, the first $k/n$ iterations place objects accessed at machine 1 at all the machines. The next $k/(n-1)$ iterations place copies of objects accessed at machine 2 at machines $2, 3, \ldots, n$, and so on, without displacing the secondary copies of objects accessed at machine 1 because of the above constraint on the probabilities of accesses. Thus, the number of iterations of the $\delta_f$ epoch algorithm before machine $n$ can place a copy of an object it accesses is $k/n + k/(n-1) + k/(n-2), \ldots, k/1 = \Omega(k \cdot log(n))$. Thus, the time taken by an iterative small epoch algorithm could be a factor of $log(n)$ more than the corresponding long epoch version for the cost of a placement computed by the former to match that computed by the latter. □

**Lemma 3**: Let $P$ be the optimal placement computed by the amortized *Fixed-$t_{fill}$* algorithm for a given epoch $T$. Let $\delta_f$ denote a tiny epoch such that for all machines $i$, $\delta_f \cdot bw(i) \geq objsize$. Then in $\frac{T}{\delta_f} \cdot h$ iterations, the $\delta_f$ epoch iterative algorithm, achieves a placement that is within a constant factor of $P$.

*Proof Outline*: The proof of this algorithm proceeds by showing that the $\delta_f$ epoch iterative algorithm achieves a placement whose cost is within a constant factor of that of the $T$ epoch placement algorithm. The basic argument is that though the iterative algorithm introduces *greedy* replicas of copies at non optimal locations, the number of greedy replicas for every copy in the $T$ epoch algorithm is at most $h$, the height of the hierarchy. The formal proof proceeds by introducing a variant of the *InitFill* algorithm known as the *Marking* algorithm that does not assign infinite benefits to initially placed copies, but never swaps them out either. It is shown that

the *Marking* algorithm in $\frac{T}{\delta_f} \cdot h$ iterations, is within a $\frac{\lambda}{\lambda-1}$ factor of the $T$ epoch placement. Finally, the $\delta_f$ iterative algorithm is shown to be within $\frac{\lambda}{\lambda-1}$ factor of the *Marking* algorithm. □

### 6.6.3 The Doubling Epoch Algorithm

Based on the intuition gathered by the discussion above we present here an algorithm that does not assume knowledge of a fill time *a priori* and that varies the epoch times in a manner such that the overall placement at any time is *close* to optimal, *i.e.* the placement sampled at any time $t$ can be proved to be within a constant factor of the optimal achievable with 1/4'th the bandwidth available to fill the caches. We first present the algorithm, perform a worst case analysis and then proceed to explain why the bound is reasonable.

**Input**: A hierarchy $H$, a set of *objsize* sized objects $\psi$, and probabilities of access $p(i, \alpha)$ of objects $i \in \psi$ at $i \in machines(H)$, and an initial placement $P_0$

**Algorithm**:

(i) Initialize epoch length $T_0 = \delta_{fill}$, where $\delta_{fill}$ is the minimum value such that for all $i \in machines(H)$, $bw(i) \cdot \delta_{fill} \geq objsize$.

(ii) for($i = 0, T_0 = \delta_{fill}$; *until done*; $i++, T_i = T_{i-1} * 2$)

Run *InitFill* algorithm for epoch length = $T_i$, with the current placement resulting from the previous run.

(iii) Goto (i)

The *until done* in the above algorithm means that the loop iteratively executes runs of *InitFill* algorithm until such time that a *change* occurs or all objects end up

getting placed at every machine that they are accessed at. A change could possibly be an object update, a demand placement, introduction of a new object or a change in its probability of access at a particular machine. Theorem 3 below bounds the worst case performance of this algorithm.

**Theorem 3**: Assume that the bandwidth to the caches is $B[\,]$. At time $t$ seconds after a change, the doubling epoch algorithm creates a placement that is within a constant factor of the optimal placement that can be achieved by a system with bandwidth $B[\,]/4$ in time $t$ with the same initial conditions. *i.e., the doubling epoch algorithm computes a placement that is within a constant factor of optimal at any instant between changes with a 4X blow-up in bandwidth.*

To prove theorem 3, we first introduce the following lemma:

**Lemma 1**: For any two given initial placement $I_1$ and $I_2$, such that $I_2$ is a superset of $I_1$, the *InitFill* algorithm starting with the initial placement $I_2$ with a fill time $T$, computes a placement $P$ that is within a constant factor of the optimal placement $P_{opt}$ obtained by starting with the placement $I_1$ and the same fill time $T$.

*Proof*: By the proof of optimality to within a constant factor of the *InitFill* algorithm, we assert that for any given epoch time, the placement $P$ computed by the *InitFill* algorithm starting with the initial placement $I_2$ is within a constant factor of the optimal placement starting with the initial placement $I_2$. However, it is also trivially true that cost of the optimal placement starting with $I_2$ is at most the cost of the optimal placement obtained by starting with $I_1$, (given the same fill epochs), since $I_2$ is given to be a superset of $I_1$. It follows that $P$ is within a constant factor of the optimal placement starting with $I_1$. □

**Proof of Theorem 3**: Starting from an initial placement $I_1$, at any time $t$ the doubling epoch algorithm has completed epochs of length $t/4, t/8, \ldots$ (and has partially

completed the epoch of length $t/2$). Denote the placement at the beginning of the epoch of length $t/4$ by $I_2$. It is clear that $I_2$ is a superset of $I_1$. Since the doubling epoch algorithm has completed an epoch of length $t/4$, the resultant placement is at least as good as the placement we would have obtained in time $t$ with 1/4'th the bandwidth and starting with $I_1$. This follows from lemma 1 above. □

Note that the bandwidth blow up of 4 is a loose worst case estimate. The proof relies only on the fact that at time $t$, we have completed a sub-epoch of length $t/4$. However, we also will have completed sub-epochs of length $t/8, t/16, \ldots$. During the *short epochs* the system will place high-benefit objects into machines that are typically close to their optimal locations. The quadrupling result of bandwidth gives no credit for this. Thus, in practice, the epoch doubling algorithm will give much better performance than a 4X blow up in bandwidth.

It must also be emphasized that a 4X increase in bandwidth may not very damaging to the hit rate. Web caches typically exhibit a log-linear relationship between cache size and hit rate. Doubling a cache's size normally increases hit rate by less than 5% (*e.g.* for web caches) [84, 174], so a 4X blow-up in bandwidth often will not hurt the hit rate much. The epoch doubling algorithm handles the problem of choosing the *optimal* epoch, since we no longer have to pick an epoch *a priori*, whatever arbitrary changes occur in the set of objects or the access pattern. Between changes we are assured to be within a constant factor of the optimal. Observe that this property implies that for a static set of objects and access probabilities, the transient access cost of the *DoublingEpoch* algorithm with a bandwidth blow-up of four is also assured to be within a constant factor of the optimal achievable.

### 6.6.4 Dynamic Continuous Placement

The *InitFill* and the *DoublingEpoch* algorithms provide a basis for coping with systems that may be changing almost continuously because of (i) object updates causing

previously placed copies to be invalidated, (ii) creation of new objects to be placed, (iii) a demand-read of an object by a cache resulting in an extra copy of an object, (iv) changes in the system's estimates of object-access probabilities, (v) changes in the system's estimate of network performance (e.g., fill bandwidths or inter-machine distances).

With the *DoublingEpoch* algorithm, whenever a change occurs, the system begins a new placement epoch with $t_{fill} = 1$ and with the updated situation as input. Unfortunately, the optimality result to within a constant factor with a $4X$ blow-up in bandwidth holds for the *DoublingEpoch* algorithm between changes, but not across changes. On one hand, if a change is *large*, resetting $t_{fill} = 1$ and *starting over* may be appropriate. On the other hand, if a change is *small*, a less radical adjustment to the schedule seems in order. Determining how to update a placement schedule so that the disruption to the original schedule is proportional to the scale of the change event is an interesting topic for future work. Some heuristics worth exploring are (i) periodic resets based on diurnal patterns of object updates, (ii) resetting when $c(P_{new}(k+1)) - c(P_{old}(k+1)) \geq \eta \cdot c(P_{old}(k+1))$, where $c(P_{new}(k+1))$ and $c(P_{old}(k+1))$ are the costs of the placements at the end of the *next* epoch, (the current epoch being the $k$'th) using the old and new object set and access probabilities resp. and $\eta < 1$ is a constant, (iii) resets based on monitoring objects updates or number of of new objects as a percentage of the current total number of objects.

Although the worst-case performance of resetting $t_{fill}$ to a small value when changes occur may be poor, this approach may still offer a reasonable heuristic because it is conservative – using too short a $t_{fill}$ interval causes the system to place *important* objects into key subtrees early (at the cost of not picking the best node within a subtree). Emperically evaluating the performance of different heuristic algorithms for varying the epoch time is an avenue for future work.

In section 6.6.2 we proved that the $\delta_f$ epoch iterative algorithm could be a factor $log(n)$ worse in the number of epochs. However, it can be shown that with a factor $h$ blow-up in the number of epochs, where $h$ is the height of the hierarchy, the $\delta_f$ epoch iterative algorithm achieves a placement that is within a constant factor of the optimal achievable. The formal statement and a proof outline are below.

**Lemma 3**: Let $P$ be the optimal placement computed by the amortized *Fixed-$t_{fill}$* algorithm for a given epoch $T$. Let $\delta_f$ denote a tiny epoch such that for all machines $i$, $\delta_f \cdot bw(i) \geq objsize$. Then in $\frac{T}{\delta_f} \cdot h$ iterations, the $\delta_f$ epoch iterative algorithm, achieves a placement that is within a constant factor of $P$.

*Proof Outline*: The proof of this algorithm proceeds by showing that the $\delta_f$ epoch iterative algorithm achieves a placement whose cost is within a constant factor of that of the $T$ epoch placement algorithm. The basic argument is that though the iterative algorithm introduces *greedy* replicas of copies at non optimal locations, the number of greedy replicas for every copy in the $T$ epoch algorithm is at most $h$, the height of the hierarchy. The formal proof proceeds by introducing a variant of the *InitFill* algorithm known as the *Marking* algorithm that does not assign infinite benefits to initially placed copies, but never swaps them out either. It is shown that the *Marking* algorithm in $\frac{T}{\delta_f} \cdot h$ iterations, is within a $\frac{\lambda}{\lambda-1}$ factor of the $T$ epoch placement. Finally, the $\delta_f$ iterative algorithm is shown to be within $\frac{\lambda}{\lambda-1}$ factor of the *Marking* algorithm. □

## 6.7 Variable Sized Objects

In this section we show that the bandwidth-constrained placement problem for variable sized objects, even for a static universe of unchanging objects and access probabilities, is a hard problem. This can be shown by a straightforward reduction from the Knapsack problem. However it is still tempting to look for approximation algorithms by tweaking the above algorithms appropriately. However, in the follow-

ing, we establish that unless P=NP, no polynomial time algorithm can provide a finite approximation guarentee to the bandwidth-constrained hierarchical placement problem.

To establish this, suppose there does exist an approximation algorithm $\mathcal{A}$ that produces a placement of variable sized objects to within a constant factor $c$ of the optimal, for a fixed $c$. We show that $\mathcal{A}$ can be used to solve the partition problem (which is known to be NP-complete).

The Partition problem takes as input a finite set $\mathcal{S}$ of objects with positive sizes and partitions them into two subsets $S_1$ and $S_2$ such that the sum of the sizes of objects in $S_1$ is equal to the sum of the sizes of objects in $S_2$.

Let $\mathcal{S} = \{a_1, a_2, \ldots, a_n\}$ be an instance of the partition problem. Let $s(a_i) \in Z^+$ be the size for each object, and $C = \sum_{i=1}^{n} s(a_i)$. In order to solve the partition problem using algorithm $\mathcal{A}$, consider a two level hierarchy consisting of two caches each of size $\frac{C}{2}$, connected by a root. Assume that the diameter of this hierarchy is 1 and the miss penalty is $nc$. Set the probability of access of each object at each cache to be a fixed value, say $p$. It is now straightforward to show that a partition of $\mathcal{S}$ exists if and only if $\mathcal{A}$ can produce a placement with cost less than $n$.

Analogous to the value density heuristic for the Knapsack problem, we can modify the Amortized *Fixed-$t_{fill}$* algorithm's swapping phase as follows:

**Swapping phase:** Swap out the object with the least *AssignedBenefit*-density *i.e.,* $\frac{Assignedbenefit}{size}$ and replace it with the *P-missing* object $\alpha$ with the highest *value*-density *i.e.,* $\frac{value}{size}$ that fits into the available cache space.

A theoretical analysis of the above algoritm is complicated. We intend to measure as part of future work performance of various heuristics for variable sized objects.

A good algorithm for the placement of variable sized objects immediately

allows us to efficiently solve a restricted steady-state version of the bandwidth constrained placement problem, where the universe of objects and their probabilities of access are fixed, the only *changes* in the system are object updates, and all objects are of the same size *objsize*. Let $\mu(\alpha)$ denote the frequency of update of object $\alpha$ at the server(s). Assume that in steady-state all of the bandwidth to a cache is used for keeping objects stored in the cache fresh. Thus, the bandwidth $B(\alpha)$ consumed by a copy of $\alpha$ at any machine is $\mu(\alpha) \cdot objsize$. Given the frequency of access $f(i, \alpha)$ of object $\alpha$ at machine $i$, the steady-state bandwidth constrained problem is to minimize the overall cost of access, as defined in the frequencies model developed in [111]. Thus, the steady-state bandwidth constrained placement problem can be viewed as a direct instance of a space constrained placement problem with the size of object $\alpha$ as $B(\alpha)$ and the space available at machine $i$ as $bw(i)$.

# Chapter 7

# Online Hierarchical Cooperative Caching

In this section, we address a hierarchical generalization of the well-known disk paging problem. In the hierarchical cooperative caching problem, a set of $n$ machines residing in an ultrametric space cooperate with one another to satisfy a sequence of read requests to a collection of (read-only) files. A seminal result in the area of competitive analysis states that LRU (the widely-used deterministic online paging algorithm based on the "least recently used" eviction policy) is constant-competitive if it is given a constant-factor blowup in capacity over the offline algorithm. Does such a constant-competitive deterministic algorithm (with a constant-factor blowup in the machine capacities) exist for the hierarchical cooperative caching problem? The main contribution of the present paper is to answer this question in the negative. More specifically, we establish an $\Omega(\log \log n)$ lower bound on the competitive ratio of any online hierarchical cooperative caching algorithm with capacity blowup $O((\log n)^{1-\varepsilon})$, where $\varepsilon$ denotes an arbitrarily small positive constant.

## 7.1 Introduction

The traditional *paging* problem, which has been extensively studied, is defined as follows. Given a cache and a sequence of requests for files of uniform sizes, a system has to satisfy the requests one by one. If the file $f$ being requested is in the cache, then no cost is incurred; otherwise a uniform retrieval cost is incurred to place $f$ in the cache. If need be, some files, determined by an online caching algorithm that does not know the future request sequence, are evicted to make room for $f$. The objective is to minimize the total retrieval cost by wisely choosing which files to evict. The cost of the online algorithm is compared against that of an optimal offline algorithm (OPT) that has full knowledge of the request sequence. Following Sleator and Tarjan [162], we call an online algorithm $c$-competitive if its cost is at most $c$ times that of OPT for any request sequence. It is well-known that an optimal offline strategy is to evict the file that will be requested furthest in the future.

The paging problem is also known as *caching* if the files have nonuniform size and retrieval cost. In their seminal paper, Sleator and Tarjan [162] have shown that LRU (Least-Recently-Used) and several other deterministic paging algorithms are $\frac{k}{k-h+1}$-competitive, where $k$ is the cache space used by LRU and $h$ is that used by OPT. They have also shown that $\frac{k}{k-h+1}$ is the best possible among all deterministic algorithms. We call $\frac{k}{h}$ the *capacity blowup* of LRU. For files of nonuniform size and retrieval cost, Young [187] has proposed the LANDLORD algorithm and shown that LANDLORD is $\frac{k}{k-h+1}$-competitive. As stated in [187], the focus of LANDLORD "is on simple *local* caching strategies, rather than distributed strategies in which caches cooperate to cache pages across a network".

In cooperative caching [58], a set of caches cooperate in serving requests for each other and in making caching decisions. The benefits of cooperative caching have been supported by several studies. For example, the Harvest cache [26] introduce the notion of a hierarchical arrangements of caches. Harvest uses the In-

ternet Cache Protocol [180] to support discovery and retrieval of documents from other caches. The Harvest project later became the public domain Squid cache system [179]. Adaptive Web Caching [191] builds a mesh of overlapping multicast trees; the popular files are pulled down towards their users from their origin server. In local-area network environments, the xFS system [10] utilizes cooperative caches to obtain a serverless file system.

A cooperative caching scheme can be roughly divided into three components: *placement*, which determines where to place copies of files, *search*, which directs each request to an appropriate copy of the requested file, and *consistency*, which maintains the desired level of consistency among the various copies of a file. In this paper, we study the placement problem, and we assume that a separate mechanism enables a cache to locate a nearest copy of a file, free of cost, and we assume that files are read-only (i.e., copies of a file are always consistent). We focus on a class of networks called *hierarchical networks*, the precise definition of which is given in Section 7.2, and we call the cooperative caching problem in such networks the *hierarchical cooperative caching (HCC) problem*.

Our notion of a hierarchical network is constant-factor related to the notion of hierarchically well-separated tree metrics, as introduced by Bartal [18]. Refining earlier results by Bartal [18], Fakcharoenphol *et al.* [68] have shown that any metric space can be approximated by well-separated tree metrics with a logarithmic distortion. Hence, many results for tree metrics imply corresponding results for arbitrary metric spaces with an additional logarithmic factor.

If the access frequency of each file at each cache is known in advance, Korupolu *et al.* [112] have provided both exact and approximation algorithms that minimize the average retrieval cost. In practice, such access frequencies are often unknown or are too expensive to track. Since LRU and LANDLORD provide constant competitiveness for a single cache, it is natural to ask whether there exists a

deterministic constant-competitive algorithm (with constant capacity blowup) for the hierarchical cooperative caching problem.

In this paper, we answer this question in the negative. We show that $\Omega(\log\log n)$ is a lower bound on the competitive ratio of any deterministic online algorithm with capacity blowup $O((\log n)^{1-\varepsilon})$, where $n$ is the number of caches in the hierarchy and $\varepsilon$ is an arbitrarily small positive constant. In particular, we construct a hierarchy with a sufficiently large depth and show that an adversary can generate an arbitrarily long request sequence such that the online algorithm incurs a cost $\Omega(\log\log n)$ times that of the adversary. Interestingly, the offline algorithms associated with our lower bound argument do not replicate files.

On the other hand, if an online algorithm is given a sufficiently large capacity blowup, then constant competitiveness can be easily achieved. Appendix 7.6 shows a simple result that, given $(1+\varepsilon')d$ capacity blowup, where $d$ is the depth of the hierarchy (i.e., $d = \Theta(\log n)$) and $\varepsilon'$ an arbitrarily small positive constant, a simple LRU-like online algorithm is constant-competitive. Note that in terms of $d$, our lower bound result yields that if the capacity blow up is $O(d^{1-\varepsilon})$, then the competitive ratio is $\Omega(\log d)$. Hence, our results imply that there is a very small range of values of the capacity blowup that separates the regions where constant competitiveness is achievable and unachievable.

Drawing an analogy to traditional caching, where LRU and LANDLORD provide constant competitiveness, we may think that a constant-competitive algorithm exists for HCC, being perhaps a hierarchical variant of LRU or LANDLORD. In fact, we began our investigation by searching for such an algorithm. Since the HCC problem generalizes the paging problem, we cannot hope to achieve constant competiveness without at least a constant capacity blowup. (In this regard, we remark that the results of [112] are incomparable as they do not require a capacity blowup.)

Several paging problems (e.g., distributed paging, file migration, and file allo-

cation) have been considered in the literature, some of which are related to the HCC problem. (See, e.g., the survey paper by Bartal [19] for the definitions of these problems.) In particular, the HCC problem can be formulated as the read-only version of the distributed paging problem on ultrametrics. And the HCC problem without replication is a special case of the constrained file migration problem where accessing and migrating a file has the same cost. Most existing work on these problems focuses on upper bound results, and lower bound results only apply to algorithms without a capacity blowup. For example, for the distributed paging problem, Awerbuch *et al.* [13] have shown that, given polylog$(n, \Delta)$ capacity blowup, there exists deterministic polylog$(n, \Delta)$-competitive algorithms on general networks, where $\Delta$ is the normalized diameter of the network. For the constrained file migration problem, Bartal [18] has given a deterministic upper bound of $\Omega(m)$, where $m$ is the total size of the caches, and a randomized lower bound of $\Omega(\log m)$ in some network topology, and an $O(\log m \log^2 n)$ randomized upper bound for arbitrary network topologies. Using the recent result of Fakcharoenphol *et al.* [68], the last upper bound can be improved to $O(\log m \log n)$.

The rest of this paper is organized as follows. Section 7.2 gives the preliminaries of the problem. Sections 7.3 and 7.4 present the main result of our paper, a lower bound for constant capacity blowup. Section 7.5 provides some concluding remarks. Appendix 7.6 presents an upper bound for sufficiently large capacity blowup.

## 7.2 Preliminaries

In this section we formally define the HCC problem. We are given a fixed six-tuple

$$(\mathcal{F}, \mathcal{C}, \mathit{dist}, \mathit{size}, \mathit{cap}, \mathit{penalty}),$$

where $\mathcal{F}$ is a set of files, $\mathcal{C}$ a set of caches, *dist* a function from $\mathcal{C} \times \mathcal{C}$ to $\mathbf{N}$, *size* a function from $\mathcal{F}$ to $\mathbf{N}$, *cap* a function from $\mathcal{C}$ to $\mathbf{N}$, *penalty* a function from $\mathcal{F}$ to $\mathbf{N}$, and $\mathbf{N}$ denotes nonnegative integers. We assume that *dist* is an ultrametric (defined below) over $\mathcal{C}$, and we assume that for every file $f$ in $\mathcal{F}$, $penalty(f) \geq diam(\mathcal{C})$, where $diam(U)$ denotes $\max_{u,v \in U} dist(u,v)$ for every set of caches $U$.

### 7.2.1 Ultrametrics and Hierarchical Networks

A distance function $d : \mathcal{C} \times \mathcal{C} \rightarrow \mathbf{N}$ is defined to be a *metric* if $d$ is nonnegative, symmetric, satisfies the triangle inequality, and $d(u,v) = 0$ if and only if $u = v$. An *ultrametric* is a special case of a metric that satisfy the inequality $d(u,v) \leq \max(d(u,w), d(v,w))$, which subsumes the triangle inequality $d(u,w) \leq d(u,v) + d(v,w)$.

An equivalent and perhaps more intuitive characterization of our ultrametric assumption is that the caches in $\mathcal{C}$ form a "hierarchical tree", or simply, a tree defined as follows. Every leaf node of the tree corresponds to a (distinct) cache. Every node in the tree has an associated nonnegative value, called the *diameter* of the node, such that for every two caches $u$ and $v$, $dist(u,v)$ equals the diameter of the least common ancestor of $u$ and $v$.

Since a hierarchical network has a natural correspondence to a tree, in the rest of this paper, we use tree terminology to develop our algorithms and analysis. In what follows, the definitions of ancestor, descendant, parent, and children follow the standard tree terminology. We use $T$ to denote the tree of caches and we use *root* to denote the root of $T$. The *depth* of *root* is 0, and the depth of $T$ is the maximum depth of any of its nodes. The *capacity* of a node is the total capacity of all the caches within the subtree rooted at that node. We impose an arbitrary order on the children of every internal node.

### 7.2.2 The HCC Problem

The goal of an HCC algorithm is to minimize the total cost incurred in the movement of files to serve a sequence of requests while respecting capacity constraints at each cache. To facilitate a formal definition of the problem, we introduce additional definitions below.

A copy is a pair $(u, f)$ where $u$ is a cache and $f$ is a file. A set of copies is called a *placement*. If $(u, f)$ belongs to a placement $\mathcal{P}$, we say that a copy of $f$ is placed at $u$ in $\mathcal{P}$. A placement $\mathcal{P}$ is *b-feasible* if the total size of the files placed in any cache is at most $b$ times the capacity of the cache. A 1-feasible placement is simply referred to as a *feasible* placement.

Given a placement $\mathcal{P}$, upon a request for a file $f$ at a cache $u$, an algorithm incurs an access cost to serve the request. If $\mathcal{P}$ places at least one copy of $f$ in any of the caches, then the cost is defined to be $size(f) \cdot dist(u, v)$, where $v$ is the closest cache at which a copy of $f$ is placed; otherwise the cost is defined to be $penalty(f)$. After serving a request, an algorithm may modify its placement via an arbitrarily long sequence of the following two operations: (1) it may add any copy to $\mathcal{P}$ and incur an access cost as defined above, or (2) it may remove any copy from $\mathcal{P}$ and incur no cost.

Given a capacity blowup of $b$, the goal of an HCC algorithm is to maintain a $b$-feasible placement such that the total cost is minimized.

## 7.3 The Lower Bound

In this section, we show that, given any constant capacity blowup $b$, the competitive ratio of any online HCC algorithm is $\Omega(\log d)$, where $d$ is the depth of the hierarchy. We prove this lower bound algorithm by showing the existence of a suitable hierarchy, a set of files, a request sequence, and a feasible offline HCC algorithm that incurs

an $\Omega(\log d)$ factor lower cost for that request sequence than any online $b$-feasible HCC algorithm. This result easily extends to analyzing how the lower bound on the competitive ratio varies as a function of nonconstant capacity blowup up to the depth of the hierarchy. In particular, with a capacity blowup of $d^{1-\varepsilon}$ for a fixed $\varepsilon > 0$, the competitive ratio of any online HCC algorithm is still $\Omega(\log d)$.

We present an adversarial argument for the lower bound. Let ON denote a $b$-feasible online HCC algorithm and ADV an adversarial offline feasible HCC algorithm. ON chooses a fixed value for the capacity blowup $b$, and ADV subsequently chooses an instance of an HCC problem (i.e., the six-tuple as introduced in Section 7.2) as follows. The hierarchy *root* consists of $n$ unit-sized caches that form the leaves of a regular $k$-ary tree with depth $d = 4bk$. Thus, for a given choice of $k$, $n = k^{4bk}$. The set of files $\Psi$ consists of $\Theta(\frac{n}{k})$ unit-sized files. The diameter of each hierarchy at depth $4bk - 1$ is 1, and the diameter of every non-trivial hierarchy is at least $\lambda$ times the diameter of any child, where $\lambda > 1$. For any file $f$, $penalty(f)$ is at least $\lambda \cdot diam(root)$. Given an instance of an HCC problem as described in Section 7.2, we give a program that takes ON as an input and generates a request sequence and a family of offline HCC algorithms each of which incurs a factor $\Omega(\log d)$ less cost than ON. We use the name OFF to refer to one algorithm in this family.

At a high level, ON's lack of future knowledge empowers ADV to play a game analogous to a *shell game*[1]. In this game, OFF maintains a compact placement of files tailored for the request sequence that ADV generates, while ON is forced to guess OFF's placement and incurs relocation costs if it guesses incorrectly. When ON finally zeroes in on OFF's placement, OFF switches its placement around, incurring a small fraction of the relocation cost that ON has already expended, and repeats the game.

As an example, consider a simple two-level hierarchy associated with equal-

[1]Thimblerig played especially with three walnut shells.

sized departments within a university. A set of files, say $A$, are of university-wide interest, while the remaining files are of department-specific interest. The capacity constraints are set up in such a way that a department can either cache files of its interest or of the university's, but not both sets simultaneously. OFF stores all the files in $A$ in an "idle" department, i.e., one with no access activity. On the other hand, ON has to guess the identity of the idle department. If ON guesses incorrectly, ADV creates requests that force ON to move files in $A$ to a different department. The best strategy for ON is to evenly distribute files in $A$ across all departments that have not yet been exposed as nonidle. Unfortunately, even with this strategy ON ends up incurring a significantly higher cost than OFF. Of course, in this simplistic case, ON can circumvent its predicament simply by a two-fold blowup in capacity and using the algorithm described in the Appendix 7.6. In the rest of the paper, we present a formalization of the shell-game-like adversarial strategy and an extension of this strategy to hierarchies of nonconstant depth.

### 7.3.1 The Adversary Algorithm ADV

We fix $d+1$ disjoint sets of files $S_0, S_1, \ldots, S_d$ such that $|S_d| = 1$ and $|S_i| = k^{d-i-1}$ for all $0 \leq i < d$. We call $i$ the depth of a file $f$ if $f \in S_i$. We define the function $g(i,j)$, where $i \geq 0$ and $j > 0$, as

$$g(i,j) = k^{d-i} \cdot \left( \frac{i-1}{4k} + \frac{1}{2j} \right).$$

ADV is shown in Figure 7.1 and the key notations used in the algorithm (and the rest of the paper) are explained in Table 7.3.1. In ADV, the nonnegative integer $N$ specifies the number of requests to be generated. The code in Figure 7.1 only shows how ADV generates a bad request sequence for ON. In Section 7.4, we show how to augment this code to obtain an offline algorithm that serves the same request sequence but incurs a much lower cost.

| Notation | Meaning |
|---|---|
| $\alpha.parent$ | the parent of $\alpha$ |
| $\alpha.anc$ | the ancestors of $\alpha$ |
| $\alpha.desc$ | the descendants of $\alpha$ |
| $\alpha.depth$ | the depth of $\alpha$ |
| $\alpha.diam$ | the diameter of $\alpha$ |
| $\alpha.files$ | $S_i$, where $i = \alpha.depth$ |
| $\alpha.cap$ | the total capacity of the caches in $\alpha$ |
| $\alpha.ch$ | children hierarchies of $\alpha$ |
| $\alpha.placed$ | the set of (distinct) files placed in the caches in $\alpha$ |
| $\alpha.load$ | the number of files $f$ in $\alpha.placed$ such that the depth of $f$ is less than $\alpha.depth$ |
| $\alpha.missing$ | the set of files $f$ such that the depth of $f$ is $\alpha.depth$ but $f \notin \alpha.placed$ |
| $\alpha.act$ | $g(\alpha.depth, r)$, where $r = \lvert\{\beta : \beta \in \alpha.parent.ch : \beta.x = 0\}\rvert$, the "activation" value |
| $\alpha.react$ | $g(\alpha.depth, k)$, the "reactivation" value |
| $\alpha.deact$ | $g(\alpha.depth, 2k)$, the "deactivation" value |

Table 7.1: Key notations.

For every node, ADV maintains two integer fields, $x$ and $y$, to summarize the state of ON. In ADV, $\pi$ is a global variable that records the current node where ADV generates the next request. Initially, $\pi$ is set to *root*. The program proceeds in rounds. At the end of each round, the algorithm generates a request. Based on ON's adjustment of its own placement, ADV adjusts $\pi$ using the up loop and the down loop. The former moves $\pi$ to an ancestor while the latter moves it to a descendant.

### 7.3.2 Correctness of ADV

We show in this section that ADV is well-defined (i.e., $\pi \neq root$ just before line 12, $\pi$ is not a leaf just before line 8, and line 14 finds a child) and that each round terminates with the generation of a request. For the sake of brevity, in our reasoning below, we call a predicate a *global invariant* if it holds everywhere in ADV (i.e., it holds initially and it holds between any two adjacent lines of the pseudocode in

```
    {initially, N ≥ 0, count = 0, π = root, root.x = root.y = root.act = g(0,k), and α.x = α.y = 0 for
 1  while count < N do {main loop}
 2     while π.load < π.deact do {up loop}
 3        π.y := π.react;
 4        for every child δ of π, set both δ.x and δ.y to 0;
 5        π := π.parent
 6     od; {end of up loop}
 7     while π.missing = ∅ do {down loop}
 8        if a child δ of π satisfies δ.x > 0 ∧ δ.load ≥ δ.react then
 9           π := δ
10        else
11           if π has exactly one child with x equal to 0 then
12              for every child δ of π, set both δ.x and δ.y to 0
13           fi;
14           π := a child δ of π such that δ.x = 0 ∧ δ.load ≥ δ.act;
15           set both π.x and π.y to π.act
16        fi
17     od; {end of down loop}
18     generate a request for an element in π.missing at an arbitrary cache in π;
19     ON serves the request and arbitrarily updates its placement;
20     count := count + 1
21  od {end of main loop}
```

Figure 7.1: The ADV algorithm.

Figure 7.1).

**Lemma 7.3.1** *Let $I_1$ denote that every internal node has a child with the $x$ field equal to 0, $I_2$ denote that $\pi$ is an internal node, and $I_3$ denote that $\pi.load \geq \pi.deact$. Then $I_1 \wedge I_2$ is a global invariant and $I_3$ holds everywhere in the down loop.*

*Proof:* The predicate $I_1 \wedge I_2$ holds initially because $\pi = root$ and $\alpha.x = 0$ for all $\alpha$, and $I_3$ holds just before the down loop due to the guard of the up loop. We next show that every line of code out of the down loop preserves $I_1 \wedge I_2$ (i.e., if $I_1 \wedge I_2$ holds before the line, then it holds after the line) and every line of code in the down loop preserves $I_1 \wedge I_2 \wedge I_3$.

Every line of code out of the down loop preserves $I_1$ because none assigns a nonzero value to a $x$ field. The only line that affects $I_2$ is line 5. We observe that $\pi \neq root$ just before line 5, due to the guard of the up loop and the observation that $root.load \geq root.deact = 0$. Hence, line 5 preserves $I_2$.

In the down loop, the only line that affects $I_1$ is 15, but $I_3$ and the inner **if** statement establish that $\pi$ has at least two children with the $x$ field equal to 0 just before line 14. Hence, line 15 preserves $I_1$. The only lines that affect $I_2$ are lines 9 and 14. We first observe that just before line 8, $\pi.depth < 4bk - 1$. This is because $I_2$ states that $\pi.depth < 4bk$ and $I_3$ implies that if $\pi.depth = 4bk - 1$, then $\pi.load \geq \pi.deact = bk - \frac{1}{4}$. Since $\pi.load$ is an integer, this implies that $\pi.load \geq bk$, which implies that $\pi.missing \supseteq S_{4bk} \neq \emptyset$, a contradiction to the guard of the down loop. Hence, $\pi.depth < 4bk - 1$ just before line 8. Therefore, line 9 preserves $I_2$. We now show that line 14 also preserves $I_2$. Let $A = \{\alpha : \alpha \in \pi.ch \wedge \alpha.x = 0\}$ and $B = \{\beta : \beta \in \pi.ch \wedge \beta.x > 0\}$. Let $r$ denote $|A|$ and $i$ denote $\pi.depth$. We observe

that

$$\begin{aligned}
& \sum_{\alpha \in A} \alpha.load \\
= & \sum_{\alpha \in A} \alpha.load + \sum_{\beta \in B} \beta.load - \sum_{\beta \in B} \beta.load \\
= & \pi.load + |S_i| - \sum_{\beta \in B} \beta.load \\
\geq & \pi.deact + |S_i| - \sum_{\beta \in B} \beta.react \\
= & g(i, 2k) + k^{d-i-1} - \sum_{\beta \in B} g(i+1, k) \\
= & r \cdot k^{d-i-1} \cdot \left( \frac{i}{4k} + \frac{1}{2r} + \frac{1}{2k} \right).
\end{aligned}$$

(In the derivation above, the second equality is due to the guard of the down loop and the definition of *load*, and the first inequality is due to the guard of the outer **if** statement.) Hence, by an averaging argument, there exists a child $\delta$ of $\pi$ such that

$$\begin{aligned}
& \delta.load \\
\geq & k^{d-i-1} \cdot \left( \frac{i}{4k} + \frac{1}{2r} \right) \\
= & \delta.act.
\end{aligned}$$

Hence, step 14 finds a child. And as shown above, $\pi.depth < 4bk - 1$ just before line 8. Hence, line 14 preserves $I_2$. The only lines that affect $I_3$ are 9 and 14. Both of these lines preserve $I_3$ because by definition, $\alpha.act \geq \alpha.deact$ and $\alpha.react \geq \alpha.deact$ for all $\alpha$.

The claim of the lemma then follows. □

**Lemma 7.3.2** *The up loop terminates.*

*Proof:* Every iteration of the up loop moves $\pi$ to its parent, and $root.load \geq root.deact$ by definition. Hence, the up loop terminates. □

**Lemma 7.3.3** *The down loop terminates.*

*Proof:* Every iteration of the down loop moves $\pi$ to one of its children. By $I_2$ of Lemma 7.3.1, $\pi$ is always an internal node. Hence, the down loop terminates. □

**Lemma 7.3.4** *ADV terminates after generating a sequence of $N$ requests.*

*Proof:* Follows from Lemmas 7.3.2 and 7.3.3. □

## 7.4 Cost Accounting

In this section, we show that there exists an offline HCC algorithm OFF that serves the sequence of requests generated by ADV and incurs a cost that is a factor $\Omega\left(\log \frac{d}{b}\right)$ less than that incurred by any $b$-feasible online HCC algorithm.

### 7.4.1 Some Properties of ADV

We first prove some properties of ADV that follow directly from its structure. For the sake of brevity, for a property that is a global invariant, we sometimes only state the property but omit stating that the property holds everywhere.

**Lemma 7.4.1** *For all $\alpha$, $\alpha.x = 0$ or $\alpha.x \geq \alpha.react$.*

*Proof:* The claim holds initially because $\alpha.x = 0$ for all $\alpha$. The only line that assigns a nonzero value to $x$ is 15, which preserves the claim because by definition, $\alpha.act \geq \alpha.react$ for all $\alpha$. □

**Lemma 7.4.2** *For all $\alpha$, $\alpha.y$ equals 0 or $\alpha.react$ or $\alpha.x$.*

*Proof:* The claim holds initially because $\alpha.y = 0$ for all $\alpha$. The only lines that modify $x$ are 4, 12, and 15. The only lines that modify $y$ are 3, 4, 12, and 15. By inspection of the code, all of these lines trivially preserve the claim. □

**Lemma 7.4.3** *Let $P$ denote the predicate that every node in $\pi.anc$ has a positive $x$ value and every node that is neither in $\pi.anc$ nor a child of a node in $\pi.anc$ has a zero $x$ value. Then $P$ is a loop invariant of the up loop, the down loop, and the main loop.*

*Proof:* Let $A$ denote $\pi.anc$ and let $B$ denote the set of nodes that are neither in $A$ nor children of the nodes in $A$.

Every iteration of the up loop moves $\pi$ to its parent. To avoid confusion, we use $\pi$ to denote the old node (i.e., child) and $\pi'$ to denote the new node (i.e., parent). An iteration of the up loop removes $\pi$ from $A$, adds $\pi.ch$ to $B$, and sets the $x$ value of $\pi.ch$ to 0. Therefore, it preserves $P$.

Every iteration of the down loop moves $\pi$ to one of its children. To avoid confusion, we use $\pi$ to denote the old node (i.e., parent) and $\pi'$ to denote the new node (i.e., child). Suppose the down loop takes the first branch of the outer **if** statement. Then it adds $\pi'$, which has a positive $x$ value, to $A$ and removes $\pi'.ch$ from $B$. Hence it preserves $P$. Suppose the down loop takes the second branch of the outer **if** statement. If line 12 is executed, $P$ is preserved because line 12 preserves both $A$ and $B$ and only changes the $x$ value of the nodes in neither $A$ nor $B$. Then lines 14 and 15 preserves $P$ because they add $\pi'$, which has a positive $x$ value after line 15, to $A$ and removes $\pi'.ch$ from $B$. Hence, it preserves $P$.

The main loop preserves $P$ because both the up loop and the down loop preserve $P$. □

**Lemma 7.4.4** *For all $\alpha$, $\alpha.y \leq \alpha.x$.*

*Proof:* The claim holds initially because $\alpha.x = \alpha.y = 0$ for all $\alpha$. The only lines that modify the $x$ or $y$ field are 3, 4, 12, and 15. At lines 4, 12, and 15, the $x$ and $y$ fields become the same value. It follows from Lemma 7.4.3 and the guard of the up loop

that just before line 3, $\pi \neq root$ and $\pi.x > 0$. It then follows from Lemmas 7.4.1 and 7.4.2 that line 3 preserves $\pi.y \leq \pi.x$. □

We now introduce the notion of an active sequence to facilitate our subsequent proofs. A sequence $\langle a_0, a_1, \ldots, a_r \rangle$, where $0 \leq r < k$, is called *i-active* if $a_j = g(i+1, k-j)$ for all $0 \leq j \leq r$.

**Lemma 7.4.5** *For every internal node $\alpha$, the nonzero $x$ fields of the children of $\alpha$ form an $i$-active sequence, where $i = \alpha.depth$.*

*Proof:* The claim holds initially because $\alpha.x = 0$ for all $\alpha$. The only lines that modify the $x$ field are 4, 12, and 15. Lines 4 and 12 preserve the claim because the $x$ fields of the children of $\pi$ all become 0. Line 15 preserves the claim (for $\pi.parent$) because $\pi.x$ becomes $\pi.act$, which by definition equals $g(i+1, k-j)$, where $i = \pi.parent.depth$ and $j$ equals the number of children of $\pi.parent$ that have a positive $x$ field. □

**Lemma 7.4.6** *Let $P(\alpha)$ denote the predicate that for all $\beta$ that are not ancestors of $\alpha$, $\beta.y \leq \beta.react$. Then $P(\pi)$ holds initially and $P(\pi)$ is a loop invariant of the up loop, the down loop, and the main loop.*

*Proof:* The predicate $P(\pi)$ holds initially because $\pi = root$ and $\alpha.y = 0$ for all $\alpha$. The up loop preserves $P(\pi)$ because every iteration first establishes $\pi.y = \pi.react$ and then moves $\pi$ to its parent. The down loop preserves $P(\pi)$ because it does not set the $y$ field to a nonzero value. The main loop preserves $P(\pi)$ because both the up loop and the down loop preserve $P(\pi)$. □

### 7.4.2 Colorings

In order to facilitate the presentation of an offline algorithm in Section 7.4.3, we introduce the notion of colorings in this section and the notion of consistent place-

ments in the next.

A *coloring* of $T$ (recall that $T$ is the tree of caches) is an assignment of one of the colors {white, black} to every node in $T$ so that the following rules are observed: (1) *root* is white, (2) every internal white node has exactly one black child and $k-1$ white children, and (3) the children of a black node are black. A coloring is called *consistent* (with ADV) if for every $\alpha$, if $\alpha.x > 0$, then $\alpha$ is white.

For any coloring $C$ and any pair of sibling nodes $\alpha$ and $\beta$, we define $swapc(C, \alpha, \beta)$ (swap coloring) as the coloring obtained from $C$ by exchanging the color of each node in the subtree rooted at $\alpha$ with that of the corresponding node in the subtree rooted at $\beta$. (Note that the subtrees rooted at $\alpha$ and $\beta$ have identical structure.)

### 7.4.3 Consistent Placements

A placement is *colorable* if there exists a coloring $C$ such that: (1) for each white internal node $\alpha$ of $T$, the set of files $\alpha.files$ are stored in (and fill) the caches associated with the unique black child of $\alpha$; (2) for each white leaf $\alpha$ of $T$, the (singleton) set of files $\alpha.files$ is stored in (and fill) the cache $\alpha$. Note that in the preceding definition of a colorable placement, the coloring $C$, if it exists, is unique. A placement is called *consistent* if it is colorable and the associated coloring is consistent.

For any placement $\mathcal{P}$ and any pair of siblings $\alpha$ and $\beta$, we define $swapp(\mathcal{P}, \alpha, \beta)$ (swap placement) as the placement obtained from $\mathcal{P}$ by exchanging the contents of each cache in $\alpha$ with that of the corresponding cache in $\beta$. Note that for any colorable placement $\mathcal{P}$ with associated coloring $C$ and any pair of sibling nodes $\alpha$ and $\beta$, the placement $swapp(\mathcal{P}, \alpha, \beta)$ is colorable, and its associated coloring is $swapc(C, \alpha, \beta)$.

### 7.4.4 The Offline Algorithm OFF

For every internal node $\alpha$, we maintain an additional variable $\alpha.last$ defined as follows. First, we partition the execution of the adversary algorithm into epochs

with respect to $\alpha$. The first epoch begins at the start of execution. Each subsequent epoch begins when either line 4 or line 12 is executed with $\pi = \alpha$. The variable $\alpha.last$ is updated at the start of each epoch, when it is set to the child $\beta$ of $\alpha$ for which the line 15 is executed with $\pi = \beta$ furthest in the future. (If one or more children $\beta$ of $\alpha$ are such that line 15 is never executed with $\pi = \beta$ in the future, then $\alpha.last$ is set to an arbitrary such child $\beta$.) Note that the variables $\alpha.last$ are introduced solely for the purpose of analysis and have no impact on the execution of ADV.

At any point in the execution of ADV, the values of the *last* fields determine a unique coloring, denoted by $C_{\mathrm{OFF}}$, as follows: *root* is white and the black child of each internal white node $\alpha$ is $\alpha.last$.

We define an offline algorithm OFF that maintains a placement $P_{\mathrm{OFF}}$ as follows. We initialize $P_{\mathrm{OFF}}$ to an arbitrary consistent placement with associated coloring $C_{\mathrm{OFF}}$. We update $P_{\mathrm{OFF}}$ to $swapp(P_{\mathrm{OFF}}, \alpha, \beta)$ whenever line 4 or line 12 is executed, where $\alpha$ and $\beta$ denote the values of $\pi.last$ before and after the execution of the line. The algorithm OFF uses the placement $P_{\mathrm{OFF}}$ to serve each request generated in line 18. The placement $P_{\mathrm{OFF}}$ is not updated when OFF serves a request; $P_{\mathrm{OFF}}$ is updated only at lines 4 and 12.

**Lemma 7.4.7** *Throughout the execution of ADV, $P_{\mathrm{OFF}}$ is colorable and has associated coloring $C_{\mathrm{OFF}}$.*

*Proof:* Immediate from the way $P_{\mathrm{OFF}}$ is updated whenever a *last* field is updated. □

**Lemma 7.4.8** *Execution of line 4 or line 12 preserves the consistency of $C_{\mathrm{OFF}}$.*

*Proof:* Assume that $C_{\mathrm{OFF}}$ is consistent before line 4. So $\pi$ is white in $C_{\mathrm{OFF}}$ before line 4, because by Lemma 7.4.3, $\pi.x$ is positive before line 4. By the definition of

$C_{\text{OFF}}$, before line 4, $\pi.last$ is black. Let $\alpha$ be $\pi.last$ before line 4, and let $\beta$ be $\pi.last$ after line 4. Before and after line 4, the $x$ values of the descendants of $\alpha$ are equal to 0. By Lemma 7.4.3, the $x$ values of all proper descendants of $\beta$ are equal to 0 before and after line 4. Since $\beta.x = 0$ after line 4, the $x$ values of all descendants of $\alpha$ and $\beta$ are equal to 0 after line 4. Hence, the *swapp* operation preserves the consistency of $C_{\text{OFF}}$. The same argument applies to line 12. □

**Lemma 7.4.9** *Execution of line 15 preserves the consistency of* $C_{\text{OFF}}$.

*Proof:* Assume that $C_{\text{OFF}}$ is consistent before line 15. Line 14 implies that $\pi \neq root$ just before line 15. Let $\pi'$ denote $\pi.parent$. By Lemma 7.4.3, $\pi'.x > 0$ and hence $\pi'$ is white before line 15. Therefore, by Lemma 7.4.7, $\pi'.last$ is the black child of $\pi'$.

Let $t$ denote the start of the current epoch for $\pi'$, i.e., $t$ is the most recent time at which $\pi'.last$ was assigned. Just after time $t$, the $x$ values of all children of $\pi'$ were equal to 0. By the definition of $t$, no child of $\pi'$ has been set to 0 since time $t$. By Lemma 7.3.1, every internal node has at least one child with $x$ equal to 0. Therefore, from time $t$ until after the execution of line 15, at most $k-1$ children of $\pi'$ have had their $x$ value set to a nonzero value. (Note that line 15 is the only line that sets $x$ to a nonzero value.) Thus, by the definition of *last*, $\pi'.last.x$ remains 0 after the execution of this line. Thus, $\pi'.last \neq \pi$. Since $\pi'$ is white and $\pi'.last$ is black in $C_{\text{OFF}}$, we conclude that $\pi$ is white in $C_{\text{OFF}}$. So $C_{\text{OFF}}$ remains consistent even with the additional constraint that $\pi$ is required to be white. (Note that $\pi.x$ is set to a positive value by line 15.) □

**Lemma 7.4.10** *The placement* $P_{\text{OFF}}$ *is always consistent.*

*Proof:* We observe that $C_{\text{OFF}}$ is always consistent, due to Lemmas 7.4.8 and 7.4.9, and the observation that lines 4, 12, and 15 are the only lines that can affect the consistency of $C_{\text{OFF}}$ (because they are the only lines that modify the *last* field or

the $x$ field of any node). It then follows from Lemma 7.4.7 that $P_{\mathrm{OFF}}$ is always consistent. □

### 7.4.5 A Potential Function Argument

Let ON denote an arbitrary online $b$-feasible algorithm. In this section, we use a potential function argument to show that ON is $\Omega\left(\frac{\nu}{\nu'}\right)$- competitive, where

$$\nu = \min\left(\frac{\lambda}{8}, \frac{\ln k}{4} - \frac{1}{4}\right)$$

and $\nu' = \frac{\lambda}{\lambda - 1}$. Let $T_{\mathrm{ON}}$ denote the total cost incurred by ON. Similarly, we let $T_{\mathrm{OFF}}$ denote the total cost incurred by OFF, except that we exclude from $T_{\mathrm{OFF}}$ the cost of initializing $P_{\mathrm{OFF}}$. (This initialization cost is taken into account in the proof of Theorem 1 below.) We define $\Phi$, a potential function, as:

$$\begin{aligned} \Phi \quad = \quad & \nu \cdot T_{\mathrm{OFF}} - \nu' \cdot T_{\mathrm{ON}} + \\ & \sum_{\alpha \in \pi.anc \wedge \alpha \neq root} \alpha.parent.diam \cdot \alpha.x + \\ & \sum_{\alpha \notin \pi.anc} \alpha.parent.diam \cdot (\alpha.x - \alpha.y + \alpha.load) \end{aligned} \tag{7.1}$$

For convenience of exposition, we account for the cost of moving from the empty placement to the first placement separately.

**Lemma 7.4.11** *The cost incurred by swapp$(\mathcal{P}, \alpha, \beta)$ is at most $2 \cdot k^{d-i} \cdot \alpha.parent.diam$, where $i = \alpha.depth$.*

*Proof:* The cost incurred is the cost of exchanging the files placed in $\alpha$ and $\beta$ with each other, which is at most $2 \cdot \alpha.cap \cdot \alpha.parent.diam = 2 \cdot k^{d-i} \cdot \alpha.parent.diam$. Note that $\alpha$ and $\beta$ have the same capacity. □

**Lemma 7.4.12** *The predicate $\Phi \leq 0$ is a loop invariant of the up loop.*

*Proof:* Every iteration of the up loop moves $\pi$ to its parent. To avoid confusion, we use $\pi$ to refer to the old node (i.e., child) and we use $\pi'$ to refer to the new node (i.e., parent). Consider the change in $\Phi$ in a single iteration of the up loop. ON incurs no cost in the up loop. By the definition of $\Phi$, line 3 preserves $\Phi$. By Lemma 7.4.4, line 4 does not increase $\Phi$. Let $i = \pi.depth$. By Lemma 7.4.11, after the execution of line 4, OFF incurs a cost of at most $c = 2 \cdot k^{d-i-1} \cdot \pi.diam$ to move from the current consistent marking placement to the next. Thus, the total change in $\Phi$ in an iteration is at most

$$\begin{aligned}
& \nu \cdot c - \pi'.diam \cdot (\pi.y - \pi.load) \\
\leq\ & \nu \cdot c - \pi'.diam \cdot (\pi.react - \pi.deact) \\
=\ & \nu \cdot c - \pi'.diam \cdot (g(i,k) - g(i,2k)) \\
=\ & \nu \cdot c - \pi'.diam \cdot k^{d-i-1} \cdot \frac{1}{4} \\
\leq\ & \nu \cdot c - \frac{\lambda}{8} \cdot c \\
\leq\ & 0.
\end{aligned}$$

(In the derivation above, the first inequality is due to the guard of the up loop and line 3, and the second inequality is due to the assumption that the diameters of the nodes are $\lambda$ separated.) □

**Lemma 7.4.13** *The predicate $\Phi \leq 0$ is a loop invariant of the down loop.*

*Proof:* Every iteration of the down loop moves $\pi$ to one of its children. To avoid confusion, we use $\pi$ to refer to the old node (i.e., parent) and $\pi'$ to refer to the new node (i.e., child). ON incurs no cost in the down loop. We consider the following three cases.

Suppose that the outer **if** statement takes the first branch. In this case, OFF does not incur any cost. Thus, the change in $\Phi$ is

$$\begin{aligned} & \pi.diam \cdot (\pi'.y - \pi'.load) \\ \leq\ & \pi.diam \cdot (\pi.react - \pi.react) \\ =\ & 0, \end{aligned}$$

where the inequality is due to Lemma 7.4.6 and the guard of the outer **if** statement.

Suppose that the outer **if** statement takes the second branch and that line 12 is not executed. In this case, OFF does not incur any cost. Thus, the change in $\Phi$ is

$$\begin{aligned} & \pi.diam \cdot (\pi'.y - \pi'.load) \\ \leq\ & \pi.diam \cdot (\pi'.x - \pi'.load) \\ \leq\ & 0, \end{aligned}$$

where the first inequality is due to Lemma 7.4.4 and the second inequality is due to lines 14 and 15.

Suppose that the outer **if** statement takes the second branch and that line 12 is executed. By Lemma 7.4.11, in this case, OFF incurs a cost of $c = 2 \cdot k^{d-i-1}$.

$\pi.diam$. Thus, the change in $\phi(\pi)$ due to line 12 is at most

$$\begin{aligned}
& \nu \cdot c - \pi.diam \cdot \sum_{\delta \in \pi.ch} (\delta.x - \delta.y) \\
\leq\ & \nu \cdot c - \pi.diam \cdot \sum_{\delta \in \pi.ch} (\delta.x - \delta.react) \\
=\ & \nu \cdot c - \pi.diam \cdot \sum_{j=1}^{k-1} (g(i+1, k-j) - g(i+1, k)) \\
=\ & \nu \cdot c - \pi.diam \cdot k^{d-i-1} \sum_{j=1}^{k-1} \left( \frac{1}{2(k-j)} - \frac{1}{2k} \right) \\
\leq\ & \nu \cdot c - \left( \frac{\ln k}{4} - \frac{1}{4} \right) \cdot c \\
\leq\ & 0.
\end{aligned}$$

(In the above derivation, the first inequality follows from Lemma 7.4.6 and the first equality follows from Lemma 7.4.5.) By the analysis of the previous case (i.e., the outer **if** statement takes the second branch but line 12 is not executed), lines 14 and 15 do not increase $\Phi$. Thus, every iteration of the down loop preserves $\Phi \leq 0$. $\square$

**Lemma 7.4.14** *Lines 18 to 20 preserve* $\Phi \leq 0$.

*Proof:* The guard of the down loop ensures that there exists a file in $\pi.missing$ just before line 18. Thus, ON incurs a cost at least $\pi.parent.diam \geq \lambda \cdot \pi.diam$ at line 19. OFF incurs a cost at most $\pi.diam$ because it stores all the files in $\pi.missing \subset S_i, i = \pi.depth$, in a child of $\pi$. Let $u$ be the cache where the request is generated, and let $A$ be the set of nodes on the path from $\pi$ to $u$, excluding $\pi$.

Since ON adds a file in $\pi.missing$ to $u$, the change in $\Phi$ is at most

$$\begin{aligned}
& \nu \cdot \pi.diam - \nu' \cdot \lambda \cdot \pi.diam + \sum_{\alpha \in A} \alpha.parent.diam \\
\leq\ & \pi.diam \cdot (\nu - \nu' \cdot \lambda) + \pi.diam \cdot \sum_{j \geq 0} \lambda^{-j} \\
\leq\ & \pi.diam \cdot \left( \nu - \nu' \cdot \lambda + \frac{\lambda}{\lambda - 1} \right) \\
\leq\ & 0.
\end{aligned}$$

(In the above derivation, the last inequality follows from $\nu' = \frac{\lambda}{\lambda-1} > \frac{1}{\lambda-1} + \frac{1}{8}$.) At line 19, ON is allowed to make arbitrarily many updates to its own placement. Suppose an update causes the *load* of some nodes to increase. Then by the definition of *load*, the set of nodes with an increased *load* value form a path from, say $\alpha$, to a leaf, and ON incurs a cost of at least $\alpha.parent.diam$. Let the set of nodes on this path be $B$. Since the diameters of the nodes on this path are $\lambda$ separated, the change of $\Phi$ is at most

$$\begin{aligned}
& \sum_{\beta \in B} \beta.parent.diam - \nu' \cdot \alpha.parent.diam \\
\leq\ & \alpha.parent.diam \cdot \sum_{j \geq 0} \lambda^{-j} - \nu' \cdot \alpha.parent.diam \\
=\ & \frac{\lambda}{\lambda - 1} \cdot \alpha.parent.diam - \nu' \cdot \alpha.parent.diam \\
=\ & 0.
\end{aligned}$$

The claim of the lemma then follows. □

**Theorem 1** ON *is* $\Omega\left(\frac{\nu}{\nu'}\right)$*-competitive.*

*Proof:* Initially, $\Phi = 0$. By Lemmas 7.4.12, 7.4.13, and 7.4.14, $\Phi \leq 0$ is a loop invariant of the main loop. Therefore, by Lemmas 7.4.1 and 7.4.4, $T_{\mathrm{ON}} \geq \frac{\nu}{\nu'} \cdot T_{\mathrm{OFF}}$ holds initially and is a loop invariant of the main loop. Let $c$ be the cost incurred by OFF in moving from the empty placement to the first placement. Note that

$T_{\text{ON}}$ serves every request with a cost at least 1 (because the diameter of an internal node is at least 1). Hence, given an arbitrarily long sequence of requests, $T_{\text{ON}}$ grows unbounded. Therefore, we can make $\frac{T_{\text{ON}}}{T_{\text{OFF}}+c}$ arbitrarily close to $\frac{\nu}{\nu'}$ by increasing the length $N$ of the request sequence generated by the program. □

The $\Omega\left(\log \frac{d}{b}\right)$ bound on the competitive ratio for a capacity blowup $b = d^{1-\epsilon}$, where $\epsilon > 0$, claimed in the beginning of Section 7.3, follows from $d = 4bk$ and that OFF can choose an arbitrarily large $\lambda$.

## 7.5 Discussion

Cooperative caching has in fact found its application in areas other than distributed systems. For example, in NUCA (NonUniform Cache Architecture), a switched network allows data to migrate to different cache regions according to access frequency [106]. Although NUCA only supports a single processor at the time of this writing, multiprocessor NUCA is being developed, with data replication as a possibility.

## 7.6 An Upper Bound

We show in this section that, given $2(d+1)$ capacity blowup, where $d$ is the depth of the hierarchy, a simple LRU-like algorithm, which we refer to as HLRU *(Hierarchical LRU)*, is constant competitive. For the sake of simplicity, we assume that every file has unit size and uniform miss penalty. Our result, however, can be easily extended to handle variable file sizes and nonuniform miss penalties using a method similar to LANDLORD [187].

### 7.6.1 The HLRU Algorithm

Every cache in HLRU is $2(d+1)$ times as big as the corresponding cache in OPT. HLRU divides every cache into $d+1$ equal-sized segments numbered from 0 to $d$. For a hierarchy $\alpha$, we define $\alpha.small$ to be the union of segment $\alpha.depth$ of all the caches within $\alpha$, and we define $\alpha.big$ to be the union of $\beta.small$ for all $\beta \in \alpha.desc$.

For the rest of this section, we extend the definitions of a *copy* and a *placement* (defined in Section 7.2.2) to internal nodes as well. A copy is a pair $(\alpha, f)$ where $\alpha$ is a node and $f$ is file that is stored in $\alpha.small$. A placement refers to a set of copies. The HLRU algorithm, shown in Figure 7.2, maintains a placement $\mathcal{P}$. In HLRU, a node $\alpha$ uses a variable $\alpha.ts[f]$ to keep track of the timestamp of a file $f$.

```
    {upon a request for f at (cache) α}
 1  t := now;
 2  do
 3     flag := false;
 4     P := P ∪ {(α, f)};
 5     α.ts[f] := max(α.ts[f], t);
 6     if capacity is violated at α.small then
 7        f := file with smallest nonzero α.ts[f];
 8        P := P\{(α, f)};
 9        if f ∉ α.big then
10           t := α.ts[f];
11           α.ts[f] := 0;
12           α := α.parent;
13           flag := true
14        fi
15     fi
16  while flag
```

Figure 7.2: The HLRU algorithm.

### 7.6.2 Analysis of the HLRU Algorithm

For any node $\alpha$ and file $f$, we partition time into *epochs* with respect to $\alpha$ and $f$ as follows. The first epoch begins at the start of execution, which is assumed to be at

time 1. Subsequent epochs begin whenever line 11 is executed.

We define $\alpha.ts^*[f]$ to be the time of the most recent access to file $f$ in node $\alpha$ in the current epoch with respect to node $\alpha$ and file $f$. If no such access exists, we define $\alpha.ts^*[f]$ to be 0.

For convenience of analysis, we categorize the file movements in HLRU into two types: *retrievals* and *evictions*. Upon request of a file, the HLRU algorithm first performs a retrieval (from the beginning of the code to line 5 of the first iteration of the loop) of the file from the nearest cache that has a copy. Each subsequent iteration of the loop performs an eviction (from line 6 of an iteration to line 5 of the next iteration) of a file from $\alpha.small$ to $\alpha.parent.small$ for some node $\alpha$.

**Lemma 7.6.1** *Before and after every retrieval or eviction, for any node $\alpha$ and file $f$, $f \in \alpha.big$ iff $\beta.ts[f] > 0$ for some $\beta \in \alpha.desc$.*

*Proof:* Initially, both sides of the equivalence are false. If both sides of the equivalence are false, the only event that truthifies either side is a retrieval of $f$ at a cache $u$ within $\alpha$, which in fact truthifies both sides. It remains to prove that if both sides of the equivalence are true, and if one side becomes false, then the other side becomes false.

The only event that falsifies the left side is an eviction of the last copy of $f$ in $\alpha.big$ from $\alpha.small$. Prior to this eviction, $\beta.ts[f] = 0$ for all proper descendants $\beta$ of $\alpha$ (since the equivalence holds for $\beta$) and $\alpha.ts[f] > 0$. The eviction then sets $\alpha.ts[f]$ to 0, falsifying the right side.

The only event that can falsify the right side is an eviction of $f$ from $\alpha.small$ such that, after the eviction, $f \notin \alpha.big$. (Note that eviction of $f$ from $\beta.small$, for a proper descendant $\beta$ of $\alpha$, cannot falsify the right side because such an eviction ensures $\beta.parent.ts[f] > 0$.) Thus, falsification of the right side implies falsification of the left side. □

**Lemma 7.6.2** *Before and after every retrieval or eviction, for any node $\alpha$ and file $f$,*

$$\alpha.ts^*[f] = \max_{\beta \in \alpha.desc} \beta.ts[f].$$

*Proof:* Initially, both sides of the equality are zero. By the definition of $\alpha.ts^*[f]$, the value of $\alpha.ts^*[f]$ changes from nonzero to 0 (i.e., a new epoch with respect to $\alpha$ and $f$ begins) at line 11. By the guard of the inner **if** statement, $f \notin \alpha.big$ just before line 11. Hence, by Lemma 7.6.1, $\beta.ts[f]$ is 0 for all $\beta \in \alpha.desc$.

The value $\alpha.ts^*[f]$ increases due to some access of $f$ at a cache $u$ within $\alpha$. The equality holds because the max value on the right side is at $u$.

Between the changes of $\alpha.ts^*[f]$, only eviction of $f$ from $\alpha$ can change the max (reset it to 0) on the right side of the equality. This eviction also resets $\alpha.ts^*[f]$ to 0 because a new epoch begins. □

**Lemma 7.6.3** *Before and after every retrieval or eviction, for any node $\alpha$ and file $f$, $\alpha.ts[f] \leq \alpha.ts^*[f]$. Furthermore, just after line 8, if $f \notin \alpha.big$, then $\alpha.ts[f] = \alpha.ts^*[f]$.*

*Proof:* The first claim of the lemma follows immediately from Lemma 7.6.2. For the second claim, note that we are evicting the last copy of $f$ in $\alpha.big$ from $\alpha.small$. By Lemma 7.6.1, all proper descendants $\beta$ of $\alpha$ have $\beta.ts[f] = 0$. So $\alpha.ts[f] = \alpha.ts^*[f]$ by Lemma 7.6.2. □

In what follows, for the convenience of analysis, we define *root.parent* to be a fake node that has every file, and we define *root.parent.diam* to be the uniform miss penalty.

When a file is moved from cache $u$ to $v$, for every node $\alpha$ on the path from the least common ancestor of $u$ and $v$ to $v$ excluding the former, we charge a *pseudocost* of $\alpha.parent.diam$ to node $\alpha$.

**Lemma 7.6.4** *If a file movement (between two caches) has actual cost $C$ and charges a total pseudocost of $C'$, then*

$$C \leq C' \leq \frac{\lambda}{\lambda - 1} C.$$

*Proof:* Suppose the file movement is from cache $u$ to cache $v$. Let $\alpha$ be the least common ancestor of $u$ and $v$ and let $B$ be the nodes on the path from $\alpha$ to $v$, excluding $\alpha$. Then

$$\begin{aligned} & C \\ = \; & \alpha.diam \\ \leq \; & \sum_{\beta \in B} \beta.parent.diam \\ = \; & C' \\ \leq \; & \alpha.diam \cdot \sum_{j \geq 0} \lambda^{-j} \\ = \; & \frac{\lambda}{\lambda - 1} \cdot C. \end{aligned}$$

□

For any node $\alpha$ and file $f$, we define auxiliary variables $\alpha.in[f]$ and $\alpha.out[f]$ for the purpose of our analysis. These variables are initialized to 0. We increment $\alpha.in[f]$ whenever retrieval of file $f$ charges a pseudocost to node $\alpha$. We increment $\alpha.out[f]$ whenever eviction of file $f$ charges a pseudocost to node $\alpha$.

**Lemma 7.6.5** *For any node $\alpha$, the total pseudocost charged to node $\alpha$ due to retrievals is*

$$\sum_{f} \alpha.in[f] \cdot \alpha.parent.diam.$$

*Proof:* Follows from the observation that whenever a pseudocost is charged to node $\alpha$ due to a retrieval, the pseudocost is $\alpha.parent.diam$. □

**Lemma 7.6.6** *For any node $\alpha$, the total pseudocost charged to node $\alpha$ due to an eviction is at most*

$$\sum_f \alpha.out[f] \cdot \alpha.parent.diam.$$

*Proof:* Follows from the observation that whenever a pseudocost is charged to node $\alpha$ due to an eviction, the pseudocost is at most $\alpha.parent.diam$. □

**Lemma 7.6.7** *For any node $\alpha$ and file $f$,*

$$\alpha.out[f] \leq \alpha.in[f].$$

*Proof:* We observe that if a pseudocost is charged to a node $\alpha$ as a result of a retrieval, then the retrieval truthifies $f \in \alpha.big$. Similarly, if a pseudocost is charged to node $\alpha$ as a result of an eviction, then the eviction falsifies $f \in \alpha.big$. It then follows that

$$\alpha.out[f] \leq \alpha.in[f] \leq \alpha.out[f] + 1$$

because $f \notin \alpha.big$ initially. □

**Lemma 7.6.8** *For any node $\alpha$, the set $\alpha.big$ always contains the most recently accessed $2 \cdot \alpha.cap$ files.*

*Proof:* Let $X$ denote the set of the most recently accessed $2 \cdot \alpha.cap$ files. We consider the places where a file is added to $X$ or removed from $\alpha.big$.

A file $f$ can be added to $X$ only when $f$ is requested at a cache $u$ within $\alpha$. In this case, $f$ is added to $u.small$ and is not evicted from $u.small$ because it is the most recently accessed item. Hence, $f \in \alpha.big$.

A file $f$ can be removed from $\alpha.big$ only when it is moved from $\alpha.small$ to $\alpha.parent.small$ as the result of an eviction and there is no other copy of $f$ in $\alpha.big$. This means that $f$ is chosen as the LRU item at line 7. Since $f$ is the LRU item,

there are $2 \cdot \alpha.cap$ items $g$ in $\alpha.small$ such that $\alpha.ts[f] < \alpha.ts[g] \leq \alpha.ts^*[g]$. By Lemma 7.6.3, $\alpha.ts[f] = \alpha.ts^*[f]$ just after line 8. It follows from the definition of $ts^*$ that $f \notin X$. □

In what follows, we use OPT to refer to an optimal offline algorithm.

**Lemma 7.6.9** *For any node $\alpha$, the total pseudocost due to retrievals charged to $\alpha$ by HLRU is at most twice the pseudocost charged to $\alpha$ by OPT.*

*Proof:* Fix a node $\alpha$. For OPT, we say that a request for a file $f$ at a cache within $\alpha$ results in a miss if no copy of $f$ exists at any cache within $\alpha$ at the time of the request. For HLRU, a miss occurs if no copy of $f$ is in $\alpha.big$. By Lemma 7.6.8, HLRU incurs at most as many misses as an LRU algorithm with capacity $2 \cdot \alpha.cap$ running on the subsequence of requests originating from the caches within $\alpha$. (Note that LRU misses whenever HLRU misses.) By the well-known result of Sleator-Tarjan [162], such an LRU algorithm incurs at most twice as many misses as OPT.

Note that a miss results in a pseudocost of $\alpha.parent.diam$ being charged to $\alpha$. Therefore, the total pseudocost charged to node $\alpha$ in OPT is at least the number of misses in OPT times $\alpha.parent.diam$. Furthermore, within HLRU, a pseudocost is charged to node $\alpha$ only on a miss. Therefore, the total pseudocost charged to node $\alpha$ in HLRU is at most the number of misses incurred by HLRU times $\alpha.parent.diam$. The claim of the lemma then follows. □

**Lemma 7.6.10** *For any node a, the total pseudocost due to evictions charged to $\alpha$ by HLRU is at most four times the total pseudocost charged to node $\alpha$ by OPT.*

*Proof:* Follows immediately from Lemmas 7.6.5, 7.6.6, 7.6.7, and 7.6.9. □

**Theorem 2** *HLRU is constant competitive.*

*Proof:* Follows immediately from Lemmas 7.6.4 and 7.6.10. □

# Chapter 8

# Summary

Replication, or making copies, of data and services is a fundamental building block in the design of distributed systems. With the rapid growth of distributed applications in the last few years, large-scale replicated systems, ie systems that move large amounts of information across wide-area networks, have come to be in widespread usage. Examples of such systems include the Web, caching and prefetching systems, content distribution networks, file sharing applications, distributed databases, web crawlers, edge-service architectures, backup systems among several others.

This dissertation presents mechanisms and algorithms to improve the performance, availability, consistency and service quality of large-scale replicated services. These benefits are obtained by enabling mechanisms and algorithms for aggressive speculative replication, or ASR, that refers to massive-scale replication of objects and aggressive propagation of updates on a speculative basis. ASR trades off increased expenditure of hardware resources in return for savings in human time, a worthy bargain in the light of falling costs of computing, storage, and network bandwidth.

We summarize below the main contributions of this dissertation. On the mechanism front, the main contributions of this dissertation are as follows:

1. We make a case for aggressive speculative replication as a fundamental design primitive in building large-scale replicated systems.

2. We demonstrate through prototype-based experiments how approaches relying on manually-tuned thresholds are fundamentally flawed due to their complexity, inefficiency, and the risk of system overload. We make a case for self-tuning system support for building large-scale distributed applications that involve massive replication.

3. We develop Mars, an architecture that provides self-tuning system support for performing ASR in a manner that prevents interference between speculative and regular load. We demonstrate that the Mars approach simplifies application design by not relying on manually-tuned thresholds, more efficiently utilizes resources for improving performance and availability, and is safe from the risk of overloading system components.

4. We build usable prototypes to instantiate Mars' architecture in real-world replicated systems. In particular, to prevent network interference, we develop the abstraction of a *background transfer* that does not interfere with existing regular traffic in the network. Our network transport protocol, TCP Nice, provides this abstraction in a deployable manner with the modification of the sender-side congestion control protocol only. In addition, for greater deployablity, we develop a user-level implementation of TCP Nice to obviate a kernel level installation. TCP Nice effectively provides a two-level network-wide prioritization without making changes to any routers.

5. We demonstrate the benefits of Mars' architecture through a case study of NPS, a prototype Web prefetching system that is non-interfering and easily deployable. NPS is free from the vagaries of technology trends, workloads, and estimation error that static thresholds are subject to, and can still provide

significant reductions in response times commensurate with available spare capacity.

The goal of the algorithmic half of the dissertation is to augment the mechanism contributions with appropriate policies for selection and placement of speculatively replicated objects. We study the problems of speculative replication and caching in bandwidth- and space-constrained environments for cooperative and stand-alone caches. The main algorithmic contributions of this dissertation are as follows:

1. We develop *long-term prefetching*, a strategy for determining which objects to speculatively replicate at a large cache constrained by bandwidth or storage space in order to minimize response times. Long-term prefetching and the associated object selection criterion, Goodfetch, take into account both the access rate and the update rate of an object to determine its prefetch-worth. Long-term prefetching is useful for specualatively replicating Web objects at large proxies and content distribution servers, as is demonstrated by our simulation experiments with real proxy traces.

2. Next, we extend long-term prefetching to a distributed cooperative environment. We consider the problem of how to speculatively place objects in a set of distributed cooperative caches in a hierarchical network where bandwidth to the caches is the constraint and Goodfetch values are known for every object at every cache. We develop an object placement algorithm that is shown to be within a constant factor of the optimal. We then extend this algorithm, maintaining asymptotic optimality, to more dynamic scenarios where object access patterns and the universe of objects are not fixed a priori.

3. Finally, we consider the theoretically intriguing problem of distributed cooperative caching in hierarchical networks in a more traditional space-constrained

environment where no a priori information about access patterns is given. We show a non-constant lower bound for the competitive ratio of any online hierarchical cooperative caching algorithm that is given at most a constant factor space advantage. We then present a simple extension of the LRU algorithm to hierarchical networks called HLRU, that when given a blowup of a factor equal to the depth of the hierarchy in the capacity of each cache, lies within a constant factor of an optimal algorithm that has complete knowledge of the request sequence.

## 8.1 Ongoing Work

This dissertation has opened up several avenues for future work, some of which are ongoing efforts. In Section 4.4, we outlined how to extend Mars' support for ASR to more general instances of the general replication problem, in particular, applications requiring strict consistency guarantees. Two key ideas enabled support for ASR in systems requiring strict consistency guarantees - i)separation of prediction policy from resource scheduling, and ii) separation of consistency information from update bodies. These ideas and more form the basis of our ongoing effort to develop a unified architecture for large-scale replicated systems described below.

### 8.1.1 Towards a Unified Replication Architecture

A unified replication architecture refers to a single set of mechanisms based on which a broad range of replicated systems can be built using different combinations of consistency policies, data placement policies and communication topology policies. Informally, consistency policies such as sequential [117] or causal [95] regulate how quickly newly written data is seen by reads, placement policies such as demand-caching [93, 136], replicate-all [145], or speculative replication of data that is most likely to be accessed at a location define which nodes store local copies of which data,

and topology policies such as client-server [93, 136], hierarchy [7, 133], or ad-hoc [87] define the paths along which information flows.

A core hypothesis motivating our effort of developing a unified replication architecture is that existing systems are special cases of a more fundamental underlying architecture, but that they are superficially incompatible because they embed consistency, placement, or topology policy assumptions in their replication mechanisms. For example, Bayou [168] allows arbitrary topologies for communication among nodes, but Bayou fundamentally assumes a data placement policy where all nodes store all data. Conversely, Coda s [108] more flexible placement policy allows nodes to cache the subset of data of interest to them, but Coda fundamentally assumes a restrictive client-server communications topology policy. Because of such entaglement of mechanisms and policies, when a replication system is built for a new environment, it must often be built from scratch or must modify existing mechanisms to accomodate new policy trade-offs. A unified set of flexible and orthogonal mechanisms for consistency, placement, and topology will allow independent policy choices in each of the dimensions.

As a first step towards this goal, we have developed an architecture for PRACTI replication [56] that supports (i) partial replication – the ability to replicate arbitrary subsets of data at any location, (ii) arbitrary consistency – support for a continuum of consistency guarantees from eventual consistency to linearizability, and (iii) topology independence – support for arbitrary communication patterns between nodes in the system. Partial replication ensures that the amount of storage, computing, and network bandwidth that a node consumes is proportional to the amount of data that is of its interest. Topology independence ensures epidemic-style efficient propagation of data to nodes in the system. Arbitrary consistency ensures that the architecture can support a diverse range of applications. Two ideas are fundamental to PRACTI replication – (i) separation of replication policy from

resource management mechanisms as in Mars, and (ii) separation of consistency and update information [135]. Though the work on PRACTI replication itself is beyond the scope of this dissertation, it stands as a testimony to the impact of enabling aggressive speculative replication in large-scale distributed systems.

### 8.1.2 Future Work

This dissertation has not sufficiently addressed the issue of how to distribute available spare resources across competing applications or users to maximize global utility. Moreover, it is unclear how much absolute benefit applications can actually derive in an environment where everybody is using ASR. In the case of Web prefetching for example, there might be sufficient capacity to let a small number of users prefetch and obtain reductions in response times. It is however possible that if a large number of users start prefetching, the interference between all of the background traffic generated by the massive-scale prefetching results in no user receiving any benefit. A prefetch request becomes useless if the response does not arrive before a demand request for that object arrives. Massive-scale prefetching can result in a scenario where everybody as a whole is using up a lot of spare capacity, but no single user receives enough to be able to get any real benefit.

The policy question of what a fair allocation of spare resources across different applications is needs further research. For example, it is not clear how to allocate resources between a user who is performing a large background file transfer and another who is doing Web prefetching through her browser and yet another whose computer is automatically fetching software updates in the background. A better understanding of utilities that human beings associate with different kinds of background traffic is needed. It is conceivable that multiple classes of service prioritization or pricing will be needed to ensure fair allocation of available resources for ASR-enabled applications. To satisfactorily answer the question of what addi-

tional mechanisms and policies are needed to enable fair and globally useful ASR, empirical data of human usage patterns in ASR environments is needed.

In Section 5.7, we raised the research question of how to efficiently aggregate information about access patterns in a WAN environment to enable appropriate prioritization of speculative requests in a Mars-based system. Such aggregation mechanisms need to be combined with a discovery mechanism that automatically directs a request to the "best" replica. A better understanding of these issues will enable us to realize the vision of a world where replicas of objects are automatically created where they are likely to be accessed and routing mechanisms obliviously direct requests to suitable replicas.

Massive-scale replication of content takes us closer to the ideal of near-instantaneous delivery of content, 100% perceived availability, desired levels of consistency, predictable service quality, location-independent content delivery, tolerance to natural disasters and malicious human attacks etc., all at the cost of cheap hardware resources. This dissertation is a first step toward realizing the potential benefits of massive-scale replication.

# Bibliography

[1] Abilene, Internet2.

[2] Anurag Acharya and Joel Saltz. A study of internet round-trip delay. Technical Report CS-TR-3736, University of Maryland, 1996.

[3] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate vs ipc : The end of the road for conventional microarchitectures. In *Proceedings of the Twenty-Seventh International Symposium on Computer Architecture*, 2000.

[4] Akamai, Inc. http://www.akamai.com.

[5] Akamai. Fast internet content delivery with freeflow, Nov 1999.

[6] V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing Reference Locality in the WWW. In *Proceedings of Parallel and Distributed Information Systems*, 1996.

[7] R. Alonso and M. Blaze. Dynamic hierarchical caching for large-scale distributed file systems. In *Proceedings of the Twelvth International Conference on Distributed Computing Systems*, June 1992.

[8] D. Andersen, D. Bansal, D. Curtis, S. Seshan, and H. Balakrishnan. Sys-

tem support for bandwidth management and content adaptation in internet applications. In *OSDI*, pages 213–226, 2000.

[9] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *Symposium on Operating Systems Principles*, pages 131–145, 2001.

[10] T. Anderson, M. Dahlin, J. Neefe, D. Patterson, D. Rosselli, and R. Wang. Serverless network file systems. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 109–126, 1995.

[11] Apache HTTP Server Project. http://httpd.apache.org.

[12] Mohit Aron, Peter Druschel, and Willy Zwaenepoel. Cluster reserves: a mechanism for resource management in cluster-based network servers. In *Measurement and Modeling of Computer Systems*, pages 90–101, 2000.

[13] B. Awerbuch, Y. Bartal, and A. Fiat. Distributed paging for general networks. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 574–583, January 1996.

[14] G. Banga, P. Druschel, and J.C. Mogul. Resource containers: A new facility for resource management in server systems. In *OSDI*, 1999.

[15] D. Bansal and H. Balakrishnan. Binomial Congestion Control Algorithms. In *Infocom*, 2001.

[16] P. Barford and M. Crovella. Generating representative workloads for network and server performance evaluation, 1998.

[17] Y. Bartal. On approximating arbitrary metrics by tree metrics. In *In Proceedings of the 30'th Annual ACM Symposium on Foundations of Computer Sciences*, pages 184–193, October 1996.

[18] Y. Bartal. On approximating arbitrary metrics by tree metrics. In *Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science*, pages 184–193, October 1996.

[19] Y. Bartal. Distributed paging. In A. Fiat and G. J. Woeginger, editors, *The 1996 Dagstuhl Workshop on Online Algorithms*, volume 1442 of *Lecture Notes in Computer Science*, pages 97–117. Springer, 1998.

[20] Y. Bartal. Probabilistic approximation of metric spaces and its algorithmic applications. In *In Proceedings of the 30'th Annual ACM Symposium on Theory of Computing*, pages 161–168, may 1998.

[21] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An architecture for differentiated services, 1998.

[22] R. Bless, K. Nichols, and K. Wehrle. A lower effort per-domain behavior (pdb) for differentiated services, December 2003.

[23] T. Bonald. Comparision of TCP Reno and TCP Vegas via fluid approximation. INRIA Research Report 3563, Nov 1998.

[24] C. Bouras and A. Konidaris. Web components: A concept for improving personalization and reducing user perceived latency on the world wide web. In *The 2nd International Conference on Internet Computing*, 2001.

[25] C. Bowman, P. Danzig, D. Hardy, U. Manber, and M. Schwartz. The Harvest information discovery and access system. In *Proceedings of the 2nd Intl. World Wide Web Conference*, pages 763–771, October 1994.

[26] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Schwartz. The Harvest information discovery and access system. *Computer Networks and ISDN Systems*, 28(1–2):119–125 (or 119–126??), 1995.

[27] Lawrence S. Brakmo and Larry L. Peterson. TCP vegas: End to end congestion avoidance on a global internet. *IEEE Journal on Selected Areas in Communications*, 13(8):1465–1480, 1995.

[28] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. In *Infocom*, 1999.

[29] E. Brewer. Invited Talk. In *Proceeding of PODC*, Aug 2002.

[30] B. E. Brewington and G. Cybenko. How dynamic is the web? *WWW9 / Computer Networks*, 33(1-6):257–276, 2000.

[31] Squid Web Proxy Cache. http://www.squid-cache.org.

[32] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technology and Systems*, December 1997.

[33] P. Cao, J. Zhang, and K. Beach. Active cache: Caching dynamic contents on the web. In *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware '98)*, 1998.

[34] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. In *SIGMETRICS*, 1995.

[35] Castro and Liskov. Practical byzantine fault tolerance. In *OSDI: Symposium on Operating Systems Design and Implementation*. USENIX Association, Cosponsored by IEEE TCOS and ACM SIGOPS, 1999.

[36] V. Cate. Alex - a global filesystem. In *Proceedings of the 1992 USENIX File System Workshop*, pages 1–12, May 1992.

[37] B. Chandra. Web workloads influencing disconnected service access. Master's thesis, University of Texas at Austin, May 2001.

[38] B. Chandra, M. Dahlin, L. Gao, A. Khoja, A. Razzaq, and A. Sewani. Resource management for scalable disconnected access to web services. In *WWW10*, May 2001.

[39] B. Chandra, M. Dahlin, L. Gao, and A. Nayate. End-to-end WAN Service Availability. In *USITS*, 2001.

[40] A. Chankhunthod, P. Danzig, C. Neerdaels, M. Schwartz, and K. Worrell. A hierarchical internet object cache. In *Proceedings of the USENIX Technical Conference*, pages 22–26, January 1996.

[41] M. Charikar, S. Guha, D. Shmoys, and É. Tardos. A constant-factor approximation algorithm for the $k$-median problem. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*, pages 1–10, May 1999.

[42] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centres. In *Symposium on Operating Systems Principles*, pages 103–116, 2001.

[43] X. Chen and X. Zhang. Coordinated Data Prefetching by Utilizing Reference Information. In *PAWS*, 2001.

[44] D. Cheriton and M. Gritter. TRIAD: A new next generation internet architecture. In *USITS '01*, March 2001.

[45] Chiu and Jain. Analysis of increase and decrease algorithms for congestion avoidance in computer networks. *Journal of Computer networks and ISDN*, 17(1):1–14, June 1989.

[46] J. Cho and H. Garcia-Molina. Synchronizing a database to improve freshness. In *SIGMOD*, 2000.

[47] J. Cleary and I. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 1984.

[48] K. G. Coffman and Andrew Odlyzko. The size and growth rate of the internet. http://www.firstmonday.dk/issues/issue3_10/coffman/, 1998.

[49] E. Cohen, B. Krishnamurthy, and J. Rexford. Improving End-to-End Performance of the Web Using Server Volumes and Proxy Filters. In *SIGCOMM98*, 1998.

[50] M. Crovella and P. Barford. The network effects of prefetching. In *Infocom*, 1998.

[51] Mark Crovella and Azer Bestavros. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. In *SIGMETRICS*, May 1996.

[52] C. Cunha, A. Bestavros, and M. Crovella. Characteristics of WWW Client-based Traces. Technical Report TR-95-010, Boston University, CS Department, April 1995.

[53] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. In *ACM SIGMOD International Conference on Management of Data (SIGMOD '93)*, pages 257–266, 1993.

[54] F. Dabek, F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *SOSP*, 2001.

[55] M. Dahlin. http://www.cs.utexas.edu/users/dahlin/techTrends/data/diskPrices/data, Jan 2002.

[56] M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. Practi replication for large-scale distributed systems. Technical report, Computer Sciences, UT Austin, 2004.

[57] M. D. Dahlin. http://cs.utexas.edu/~dahlin/techTrends/data/diskPrices/data.

[58] M. D. Dahlin, R. Y. Wang, T. E. Anderson, and D. A. Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 267–280, November 1994.

[59] B. Davison. Assertion: Prefetching with get is not good. Web Caching and Content Distribution Workshop, June 2001.

[60] Brian D. Davison and Vincenzo Liberatore. Pushing politely: Improving Web responsiveness one packet at a time (extended abstract). *Performance Evaluation Review*, 28(2):43–49, September 2000.

[61] Stephen E. Deering, Deborah Estrin, Dino Farinacci, Van Jacobson, Ching-Gung Liu, and Liming Wei. An architecture for wide-area multicast routing. In *SIGCOMM*, pages 126–135, London, UK, August 1994. ACM.

[62] F. Douglis, A. Feldmann, B. Krishnamurthy, and J. C. Mogul. Rate of change and other metrics: a live study of the world wide web. In *USITS'97*, 1997.

[63] R. Doyle, J. Chase, S. Gadde, and A. Vahdat. The trickle-down effect: Web caching and server request distribution, 2001.

[64] D. Duchamp. Prefetching Hyperlinks. In *USITS*, 1999.

[65] B. Duska, D. Marwood, and M. Feeley. The measured access characteristics of world-wide-web client proxy caches. In *USITS97*, Dec 1997.

[66] S. Dykes and K. A. Robbins. A viability analysis of cooperative proxy caching. In *Infocom*, 2001.

[67] Tivoli Data Exchange. http://www.tivoli.com/products/documents/datasheets/data_exchange_ds.pdf.

[68] J. Fakcharoenphol, S. Rao, and K. Talwar. A tight bound on approximating arbitrary metrics by tree metrics. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing*, pages 448–455, June 2003.

[69] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *Proceedings of the 1998 ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 254–265, August 1998.

[70] L. Fan, P. Cao, W. Lin, and Q. Jacobson. Web prefetching between low-bandwidth clients and proxies: Potential and performance, 1999.

[71] M. Feeley, W. Morgan, F. Pighin, A. Karlin, H. Levy, and C. Thekkath. Implementing global memory management in a workstation cluster. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[72] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1, June 1999.

[73] Fireclick. Netflame. http://www.fireclick.com, Last known to have existed in 2001.

[74] S. Floyd, M. Handley, J. Padhye, and J. Widmer. Equation-based congestion control for unicast applications: the extended version. Technical Report TR-00-003, ICSI, March 2000.

[75] Sally Floyd and Van Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[76] L. Gao, M. Dahlin, A. Nayate, J. Zheng, and A. Iyengar. Application specific

data replication for edge services. In *Proceedings of the International World Wide Web*, May 2003.

[77] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of Consistent, Available, Partition-tolerant web services. In *ACM SIGACT News, 33(2)*, Jun 2002.

[78] S. Glassman. A caching relay for the World Wide Web. *Computer Networks and ISDN Systems*, 27(2):165–173, 1994.

[79] P. Goyal, X. Guo, and H.M. Vin. A hierarchical cpu scheduler for multimedia operating systems. In *OSDI*, pages 107–122, October 1996.

[80] J. Gray. Distributed computing economics. Technical Report MSR-TR-2003-24, Microsoft Research, March 2003.

[81] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution, 1996.

[82] J. Gray and P. Shenoy. Rules of thumb in data engineering. In *"Proc. 16th Internat. Conference on Data Engineering"*, pages 3–12, 2000.

[83] Jim Gray and Gordon Bell. Digital immortality. *CACM*, 44(3):28–31, 2001.

[84] S. Gribble and E. Brewer. System Design Issues for Internet Middleware Services: Deductions from a Large Client Trace. In *USITS97*, Dec 1997.

[85] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable distributed data structures for internet service construction. In *OSDI*, 2002.

[86] J. Griffioen and R. Appleton. Automatic Prefetching in a WAN. In *IEEE Workshop on Advances in Parallel and Distributed Systems*, October 1993.

[87] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeir. Implementation of the Ficus Replicated File System. In *Proceedings of the Summer 1990 USENIX Conference*, pages 63–72, Summer 1990.

[88] J. Gwertzman and M. Seltzer. An analysis of geographical push-caching, 1997.

[89] J. S. Gwertzman and M. Seltzer. The case for geographical push-caching. In *HotOS*, 1995.

[90] A. Heddaya and S. Mirdad. WebWave: Globally load balanced fully distributed caching of hot published documents. In *Proceedings of 17th Intl. Conference on Distributed Computing Systems*, May 1997.

[91] Hotmail.

[92] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[93] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[94] N. Hutchison, S. Manley, M. Federwisch, G. Harris, D. Hitz, S. Kleiman, and S. O'Malley. Logical vs. physical file system backup. In *OSDI*, 1999.

[95] P. W. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. *The 10th International Conference on Distributed Computing Systems*, pages 302–309, 1990.

[96] S. Tuecke I. Foster, C. Kesselman. The anatomy of the grid: Enabling scalable virtual organizations. In *International J. Supercomputer Applications*, volume 15(3), 2001.

[97] IBM. Websphere.

[98] IMSI Net Accelerator. http://nct.digitalriver.com/fulfill/0002.3.

[99] Intel. N-tier architecture improves scalability and ease of integration.

[100] Inc. Internet Doorway. http://www.netdoor.com/info/tiert3pricing.html.

[101] Digital Island. http://www.sandpiper.net.

[102] Quinn Jacobson and Pei Cao. Potential and limits of Web prefetching between low-bandwidth clients and proxies. In *Third International WWW Caching Workshop*, 1998.

[103] V. Jacobson. Congestion avoidance and control. In *SIGCOMM88*, 1988.

[104] G Karakostas and D. Serpanos. Practical LFU implementation for Web Caching . Technical Report TR-622-00, Department of Computer Science, Princeton University, 2000.

[105] D. Karger, F. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *In Proceedings of the 29'th Annual ACM Symposium on Theory of Computing*, pages 654–663, May 1998.

[106] C. K. Kim, D. Burger, and S. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 211–222, October 2002.

[107] Tracy Kimbrel, Andrew Tomkins, R. Hugo Patterson, Brian Bershad, Pei Cao, Edward Felten, Garth Gibson, Anna R. Karlin, and Kai Li. A trace-driven comparison of algorithms for parallel prefetching and caching. In *OSDI*, pages 19–34, 1996.

[108] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. In *Thirteenth ACM Symposium on Operating Systems Principles*, volume 25, pages 213–225, Asilomar Conference Center, Pacific Grove, U.S., 1991. ACM Press.

[109] R. Kokku, P. Yalagandula, A. Venkataramani, and M. Dahlin. A non-interfering deployable web prefetching system. Technical Report TR-02-51, Computer Sciences, UT Austin, May 2002.

[110] R. Kokku, P. Yalagandula, A. Venkataramani, and M. Dahlin. Nps: A non-interfering deployable web prefetching system. In *USITS*, March 2003.

[111] M. Korupolu and M. Dahlin. Coordinated placement and replacement for large-scale distributed caches. In *Workshop On Internet Applications*, June 1999.

[112] M. R. Korupolu, C. G. Plaxton, and R. Rajaraman. Placement algorithms for hierarchical cooperative caching. *Journal of Algorithms*, 38:260–302, 2001.

[113] B. Krishnamurthy and C. Wills. Piggyback Server Invalidation for Proxy Cache Coherency. In *WWW7*, 1998.

[114] T. M. Kroeger, D. E. Long, and J. C. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *USITS*, 1997.

[115] A. Kuzmanovic and E. W. Knightly. Tcp-lp: A distributed algorithm for low priority data transfer. In *In Proceedings of IEEE INFOCOM 2003, San Francisco, CA*, April 2003.

[116] L. Lamport. Clocks and ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–565, July 1978.

[117] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *programs. IEEE Transactions on Computers*, C-28(9):690–691, 1979.

[118] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[119] W. LeFebvre. Cnn.com: Facing a world crisis, December 2001.

[120] A. Leff, J. L. Wolf, and P. S. Yu. Replication algorithms in a remote caching architecture. *IEEE Transactions on Parallel and Distributed Systems*, 4(11):1185–1204, 1993.

[121] D. Li and D. R. Cheriton. Scalable web caching of frequently updated objects using reliable multicast. In *Proceedings of USITS'99*, 1999.

[122] W.F. Lin, S.K. Reinhardt, and D. Burger. Designing a modern memory hierarchy with hardware prefetching. In *IEEE Transactions on Computers special issue on computer systems*, volume Vol.50 NO.11, November 2001.

[123] C. Liu and P. Cao. Maintaining Strong Cache Consistency in the World-Wide Web. In *Proceedings of the Seventeenth International Conference on Distributed Computing Systems*, May 1997.

[124] C. Lumb, J. Schindler, G. Ganger, D. Nagle, and E. Riedel. Towards higher disk head utilization: Extracting free bandwidth from busy disk drives. In *OSDI*, 2000.

[125] Christopher Lumb, Jiri Schindler, Gregory R. Ganger, Erik Riedel, and

David F. Nagle. Towards higher disk head utilization: Extracting "free" bandwidth from busy disk drives. In *OSDI*, 2000.

[126] Korupolu M., G. Plaxton, and R. Rajaraman. Placement algorithms for hierarchical cooperative caching. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 586–595, January 1999.

[127] B. M. Maggs, F. Meyer auf der Heide, B. Vöcking, and M. Westermann. Exploiting locality for data management in systems of limited bandwidth. In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science*, pages 284–293, October 1997.

[128] C. Maltzahn, K. Richardson, D. Grunwald, and J. Martin. On bandwidth smoothing. In *4th International Web Caching Workshop*, 1999.

[129] E. Markatos and C. Chronaki. A Top-10 Approach to Prefetching on the Web. In *INET*, 1998.

[130] J. C. Mogul. Network Behavior of a Busy Web Server and its Clients. Technical Report WRL 95/5, DEC Western Research Laboratory, Palo Alto, California, 1995.

[131] R. Morris. Tcp behavior with many flows. In *International Conference on Network Protocols*, 1997.

[132] David Mosberger and Tai Jin. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance*, pages 59—67. ACM, June 1998.

[133] D. Muntz and P. Honeyman. Multi-level caching in distributed file systems - or - your cache ain't nuthin' but trash. *Proceedings of the USENIX Winter Conference*, pages 305–313, January 1992.

[134] Naviscope. http://www.naviscope.com, Last known to have existed in 2001.

[135] A. Nayate and M. Dahlin. Transparent replication through invalidation and prefetching. Technical Report TR-03-XX, Computer Sciences, UT Austin, September 2003.

[136] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, 1988.

[137] Netscape Communications Corporation. Javascript security.

[138] The network simulator – ns-2. http://www.isi.edu/nsnam/ns.

[139] A. Odlyzko. Internet growth: Myth and reality, use and abuse. *Journal of Computer Resource Management*, pages 23–27, 2001.

[140] V. N. Padmanabhan and J. C. Mogul. Using predictive prefetching to improve World-Wide Web latency. In *SIGCOMM96*, 1996.

[141] Vivek S. Pai, Peter Druschel, and Willy Zwaenepoel. Flash: An efficient and portable Web server. In *USENIX Annual Technical Conference*, 1999.

[142] T. Palpanas. Web prefetching using partial match prediction, 1998.

[143] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *SOSP*, 1995.

[144] V. Paxson. End-to-end Routing Behavior in the Internet. In *SIGCOMM96*, 1996.

[145] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication.

In *Proceedings of the 16th ACM Symposium on Operating SystemsPrinciples (SOSP-16), Saint Malo, France*, 1997.

[146] G. Popek, R. Guy, T. Page, and J. Heidemann. Replication in the Ficus Distributed File System. In *Workshop on the Management of Replicated Data*, pages 5–10, November 1990.

[147] R. Prasad, M. Jain, and C. Dovrolis. On the effectiveness of delay-based congestion control. In *Proceedings of the Second International Workshop on Protocols for Fast Long-Distance Networks*, February 2004.

[148] M. Rabinovich, I. Rabinovich, and R. Rajaraman. Dynamic replication on the internet. Technical report, AT&T Labs – Research, April 1998.

[149] R. Rajamony and M. Elnozahy. Measuring Client-Perceived Resonse Times on the WWW. In *USITS*, 2001.

[150] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content addressable network. In *Proceedings of ACM SIGCOMM 2001*, 2001.

[151] R. Rejaie, M. Handley, and D. Estrin. RAP: An end-to-end rate-based congestion control mechanism for realtime streams in the internet. In *Infocom*, 1999.

[152] Resonate Inc.

[153] Winter Corporation Richard Winter. Intelligent enterprise. http://www.wintercorp.com/rwintercolumns/ie_9904.html, 1999.

[154] Chief Architect Robert Blumofe. The challenges of delivering content and applications on the internet, October 2002.

[155] A. Rousskov and D. Wessels. Cache digests. In *Proceedings of the 3rd Intl. WWW Caching Workshop*, June 1998. http://wwwcache.ja.net/events/workshop/.

[156] M. Roussopoulos and M. Baker. Cup: Controlled update propagation in peer to peer networks. In *Proceedings of the 2003 USENIX Annual Technical Conference*, June 2003.

[157] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *SOSP*, 2001.

[158] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the pangaea wide-area file system. In *OSDI*, 2002.

[159] Dheeraj Sanghi, Ashok K. Agrawala, Olafur Gudmundsson, and Bijendra N. Jain. Experimental assessment of end-to-end behavior on internet. In *Infocom (2)*, pages 867–874, 1993.

[160] Stefan Savage, Tom Anderson, Amit Aggarwal, David Becker, Neal Cardwell, Andy Collins, Eric Hoffman, John Snell, Amin Vahdat, Geoff Voelker, and John Zahorjan. Detour: a case for informed internet routing and transport. *IEEE Micro*, 19(1):50–59, January 1999.

[161] P. Shenoy and H. Vin. Cello: A disk scheduling framework for next-generation operating systems. In *SIGMETRICS*, 1998.

[162] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[163] Neil T. Spring, Maureen Chesire, Mark Berryman, Vivek Sahasranaman, Thomas Anderson, and Brian N. Bershad. Receiver based management of low bandwidth access links. In *Infocom*, 2000.

[164] Van Steen, F. J. Hauck, and A. S. Tanenbaum. A model for worldwide tracking of distributed objects. In *Proceedings of the 1996 Conference for Telecommunications Information Networking Architecture (TINA '96)*, pages 203–212, September 1996.

[165] I. Stoica, R. Morris, D. Karger, M. Kaashock, and H. Balakrishman. Chord: A scalable peer-to-peer lookup protocol for internet applications, 2001.

[166] Lakshminarayanan Subramanian, Ion Stoica, Hari Balakrishnan, and Randy Katz. Overqos: An overlay based architecture for enhancing internet qos. In *Proceedings of NSDI '04*, March 2004.

[167] Cheetah Software Systems.

[168] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *SOSP*, 1995.

[169] R. Tewari, M. Dahlin, H. M. Vin, and J. S. Kay. Design considerations for distributed caching on the internet. In *Proceedings of the 19th Intl. Conference on Distributed Computing Systems*, 1999. To appear.

[170] The ICAP Protocol Group. Icap the internet content adaptation protocol. Technical Report draft-opes-icap-00.txt, IETF, December 2000.

[171] A. Vahdat, M. Dahlin, T. Anderson, and A. Aggarwal. Active names: Flexible location and transport of wide-area resources. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[172] A. Venkataramani, M. Dahlin, and P. Weidmann. Bandwidth constrained placement in a WAN. In *PODC*, 2001.

[173] A. Venkataramani, R. Kokku, and M. Dahlin. TCP-Nice: A Mechanism for Background Transfers. In *OSDI*, December 2002.

[174] A. Venkataramani, P. Yalagandula, R. Kokku, S. Sharif, and M. Dahlin. Potential costs and benefits of long-term prefetching for content-distribution. *Computer Communications Journal*, 25(4):367–375, 2002.

[175] Arun Venkataramani. Technology trends.

[176] Z. Wang. *Internet QoS: Architectures and Mechanisms for Quality of Service.* Morgan Kaufmann Publishers, March 2001.

[177] Wcol.
http://shika.aist-nara.ac.jp/products/wcol/wcol.html, Last known to have existed in 2001.

[178] Matt Welsh, David E. Culler, and Eric A. Brewer. SEDA: An architecture for well-conditioned, scalable internet service. In *SOSP*, 2001.

[179] D. Wessels. Squid Internet object cache. http://squid.nlanr.net/Squid, Jan 1998.

[180] D. Wessels and K. Cla. RFC 2187: Appliation of Internet Cache Protocol, 1997.

[181] S. Williams, M. Abrams, C. R. Standridge, G. Abdulla, and E. A. Fox. Removal policies in network caches for World-Wide Web documents. In *Proceedings of the ACM SIGCOMM'96 conference*, 1996.

[182] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems*, 22(4):255–314, 1997.

[183] Yahoo.

[184] Y. Yang and S. Lam. General AIMD Congestion Control. In *ICNP*, 2000.

[185] J. Yin, L. Alvisi, M. Dahlin, and C. Lin. Using Leases to Support Server-Driven Consistency in Large-Scale Systems. In *Proceedings of the Eighteenth International Conference on Distributed Computing Systems*, May 1998.

[186] N. E. Young. On-line file caching. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 82–86, January 1998.

[187] N. E. Young. On-line file caching. *Algorithmica*, 33:371–383, 2002.

[188] H. Yu and A. Vahdat. The costs and limits of availability for replicated services. In *SOSP*, 2001.

[189] Zeus Technology.

[190] L. Zhang, S. Michel, Nguyen K., Rosenstein A., S. Floyd, and V. Jacobson. Adaptive web caching. In *Proceedings of the 3rd Internation WWW Caching Workshop*, June 1998.

[191] Lixia Zhang, Sally Floyd, and Van Jacobson. Adaptive Web Caching. In *Proceedings of the 1997 NLANR Web Cache Workshop*, 1997.

[192] Y. Zhang, V. Paxson, and S. Shenkar. The Stationarity of Internet Path Properties: Routing, Loss, and Throughput. Technical report, ICSI Center for Internet Research, May 2000.

# Vita

Arunkumar Venkataramani was born on May 4, 1978 in Chennai, India, the son of Vishwanatha Sarma Venkataramani and Chandra Venkataramani. He received the Bachelor of Technology degree in Computer Science and Engineering from the Indian Institute of of Technology at Bombay in May 1999. Thereafter, he received the Master of Sciences degree in Computer Sciences from the University of Texas at Austin in December 2000. He was awarded the J.C. Browne graduate fellowship in December 2002.

Permanent Address: C/O V. Venkataramani
E-14, Note Mudran Nagar,
BRBNMPL Township,
Mysore Pin: 570003
Karnataka State, INDIA

This dissertation was typeset with LaTeX $2_\varepsilon$[1] by the author.

[1] LaTeX $2_\varepsilon$ is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay and James A. Bednar.