# Practical and Theoretical Issues in Prefetching and Caching

Andrew Tomkins

October 7, 1997

CMU-CS-97-181

Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

*Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.*

Thesis Committee:

Merrick Furst, Computer Science Department (chair)
Avrim Blum, Computer Science Department
Garth Gibson, Computer Science Department
Daniel D. Sleator, Computer Science Department
Richard J. Lipton, Princeton University Computer Science Department



This research is sponsored by the Department of the Navy, Office of Naval Research under Contract No. N00174-96-0002, and the Defense Advanced Research Projects Agency (DARPA), Army Research Office under Contract No. DABT63-93-C-0054. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of Defense or the United States Government.

**Abstract**

This thesis has two parts, the first more practical, and the second more theoretical. The first part considers *informed prefetching and caching* in which an application provides information about its upcoming I/O accesses to the operating system, allowing the system to prefetch data and to make informed cache replacement decisions. I compare existing algorithms for this problem using trace-driven simulation, and use the results to develop a new algorithm that performs better than previous approaches, again under trace-driven simulation.

The second part considers *weighted caching*, a theoretical problem from the domain of on-line algorithms. I present an algorithm with competitive ratio $O(\log^2 k)$ on $(k+1)$-point spaces, the first poly-logarithmic ratio for this problem. I also give an almost-tight lower bound of $\Omega(\log k)$ for any weighted caching problem on at least $k+1$ points. I then show a connection between this problem and a new on-line $k$-server model in which the servers may rearrange themselves without cost during "free-time" between requests, and describe a series of results in the free-time model.

**Keywords:** Operating Systems, Storage Management, Secondary Storage, Caching, Prefetching, Trace-Driven Simulation, TIP, Algorithms, Probabalistic Algorithms, Online Algorithms, Competitive Ratio, Weighted Caching, Metrical Task Systems, Free Time, k-Server Problems

# Contents

# List of Tables

# List of Figures

## Acknowledgments

Thanks first to my advisor Merrick Furst, who (quite consciously) did what few advisors do: advised me, but let me make my own decisions. He supported my digressions into other areas of computer science, which kept eight years of grad school exciting start to finish. He spent hours teaching me everything from spectral analysis of boolean functions to how an intro should be phrased, and what shouldn't appear on a slide. Finally, he encouraged me to incorporate a systems component into my thesis: without that opportunity the thesis, and the associated job talk, would have been far weaker.

Next, thanks to Garth Gibson and Avrim Blum. At one level, they spent many hours teaching me about systems and online algorithms respectively; at another level, I had the opportunity to watch two top-notch researchers with completely different styles in action. Garth took me on as a systems novice, late in my graduate career, and provided financial, moral and intellectual support to me all the way through two papers and a thesis. Avrim spent hours discussing online algorithms with me, and demonstrated repeatedly that somewhere in the middle of the problem, there's a key insight: the right formulation, or even the right picture; and the whole idea is to answer the *simplest* question that solves your problem.

And thank you also to the other faculty who took time to help me along the way. Dick Lipton got me started in online algorithms, and got me out to Princeton for what later grew into four pleasant summers at the Matsushita Information Technology Laboratories. Danny Sleator was always willing to discuss competitive algorithms at any level of detail from the guts of the proof to the strengths and weaknesses of the competitive analysis itself. Finally, Bruce Maggs was always happy to discuss topics from network scheduling algorithms to job search issues.

I'm very grateful to the students in Garth Gibson's Parallel Data Lab, especially Hugo Patterson. As a newcomer to systems, I've been a never-ending source of braindead questions and "observations;" the PDL not only put up with it like a many-headed Job, but spent inordinate amounts of time helping me out. A problem arises, and out of the blue David Rochberg says "Well, how about if I work day and night this weekend to create trace-collection software and trace a set of applications so you'll have better results?" Incredible. And Hugo more even than Garth has taught me essentially everything I know about systems, has walked me through code, found bugs, described countless details of applications in the test suite, and put hours into co-authoring the Sigmetrics and OSDI papers we wrote.

On a similar note, the Carnegie Mellon University School of Computer

Science is unlike any other place I've seen or imagined; I have no doubt I will miss the quality and dedication of every last staff person, graduate student, and faculty member. Thanks especially to Sharon Burks, and to Terri and Patti; I don't know how it happened, but I can't say enough good things about the place.

In addition to the technical support, I've benefited from a great deal of personal support. First and foremost, Chris Colby has been the staunchest and most considerate of friends over eight years and, at best estimate, about two million revolutions of plastic. Bryan Loyall and I overlapped for much of our respective thesis crises, and spiraled into and out of insanity together — typically it began with venting about some innocuous behavior pattern, and ended with a modification to our door that nobody else found funny. Similarly, a number of wonderful grad students, faculty, frisbee players, and Pittsburgh icons have made my stay here better than I would ever have guessed. And thanks especially to two groups of old friends, one from Oakton for continued support in the face of an international diaspora, and another from MIT who fortuitously seem to be clustering in San Jose. Finally, thanks to Lisa Haverty for seeing me through it, providing escape when I could handle it, and support when I couldn't.

Most importantly, my family remains the foundation of it all, one area of support I never question.

# Chapter 1

# Introduction

*Like a man to double business bound,*
*I stand in pause where I shall first begin*

— *William Shakespeare, "Hamlet"*

This document has two distinct parts. In the first part I consider disk prefetching and disk cache management in an operating system that allows application programs to give information about their upcoming accesses. In the second part I present a theoretical analysis of randomized algorithms for *weighted caching*, a variant of the cache management problem, and use it to extend the traditional formal model by incorporating a notion of "free time" between requests. At the beginning of each of these two parts I give a full introduction motivating the particular problem, describing related work and summarizing the results. The remainder of this high-level introduction represents a roadmap and quick summary of the document.

## 1.1 Part I: Informed Caching and Prefetching

Traditional filesystems wait until an application requires data, and then generate a request to the I/O subsystem. If the requested data is not present in the buffer cache, the I/O subsystem then generates a disk access and ejects a buffer under the LRU replacement policy. Recent work on integrated prefetching and caching suggests that both fetches and evictions can be improved using "hints" from the application about upcoming elements of the request sequence. First, I/O stall time can be reduced by prefetching data. Second, buffer cache replacement decisions can combine traditional LRU information with any available application-provided information about future requests. The benefits are significant in a single-disk system, but become much more dramatic with multiple disks because hints create the potential to perform fetches on different disks simultaneously. I approach the problem from this perspective with a short discussion of the big picture, followed by a description of the chapters to follow.

The results in this part of the thesis benefit from collaboration with Garth Gibson, Hugo Patterson, and a number of other colleagues. I give precise acknowledgments as I describe each piece of collaborative work.

### 1.1.1 The Big Picture

*Storage Parallelism* in the form of disk arrays [PGK88] has been advocated as a means to address the increasing gap between processor speed and I/O bandwidth [SGM86] (the so-called "I/O bottleneck" [PGK88, Smi85]) . But many workloads consist of streams of computation interspersed with synchronous I/O calls. The program blocks for I/O from disk 1, computes again when the data arrives, and then blocks for I/O from disk 2, never benefiting from the parallelism of the disk array.

To address this problem, existing systems for *Informed Prefetching and Caching,* in which the application provides "hints" to the system about upcoming accesses, submit requests in parallel to make use of the large aggregate bandwidth of the disk array [PGG+95, CFL94b, CFKL95b, PG94]. Figure 1.1 shows the average reduction in stall time provided by the TIP2 system of Patterson, Gibson *et al.* [PGG+95], relative to Digital's OSF/1 operating system. On a large array with applications modified to provide the necessary hints, stall is reduced by approximately a factor of 6 to 17% of its original value under OSF/1. Patterson, Gibson *et al.* argued for the feasibility of hints by modifying a wide range of I/O-intensive applications to provide hints. Mowry *et al.* showed that for some workloads, the compiler can be augmented to provide hints automatically [MDK96].

So there are strong indications that applications can be written or compiled to provide hints, and there is evidence from the work cited above that hints can provide a dramatic reduction in application stall time. But prior to the work described in this thesis, existing studies compared systems that make use of application hints to systems that do not. Now that there is a strong case for hints, it is important to understand whether existing systems are using the hints well, and if not, how they should be modified. This thesis makes two contributions that are interleaved through the upcoming chapters. First, it provides detailed comparisons of all existing systems and several new algorithms, and draws conclusions about the strengths and weaknesses of each. Second, it presents a new algorithm called TIPTOE that performs substantially better than all previous approaches on some workloads, and does not perform substantially worse than any previous approach on any workload studied.

As a final note, the TIP2 system provides a factor of 6 improvement relative to traditional systems, but also incurs the large cost associated with shifting paradigms. Patter-

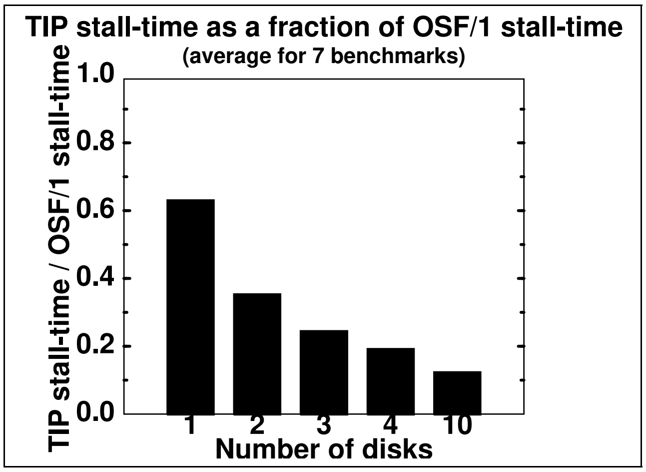



Figure 1.1: Reduction in Stall Time of Informed Prefetching and Caching versus a traditional operating system. The opportunity to hide latency increases as the disk array size increases. Over a set of benchmarks, Patterson, Gibson *et al.* note that 83% of application stall has been eliminated versus OSF/1, even though OSF/1 already includes an aggressive sequential readahead policy.

son *et al.* have argued convincingly that the dramatic improvement is worth the cost. In this thesis, I compare algorithms within the new paradigm of hint-based prefetching and cache management. Thus, I do not expect a factor of 6 difference between approaches, as all the algorithms are competing under the same rules. Typically the differences between algorithms are much more modest, perhaps 10%, with occasional workloads showing factors of 1.5–2 improvement for some approach. The goal is therefore to realize the factors of 1.5–2 versus other hint-based systems whenever possible, while consistently delivering the large improvements versus traditional systems.

### 1.1.2 Systems Roadmap

The systems part of this document evaluates various algorithms for informed prefetching and caching via trace-driven simulation. I begin in Chapter 2 with a more detailed motivation of the problem, a preliminary discussion of the results to follow, and then an overview of related work.

Next, Chapter 3 gives details about the simulator, the trace collection mechanism, the applications used to generate the traces, and the manner in which hints are given to the system. The simulator is trace-driven, provides multi-threading for simultaneous playback of multiple traces, and contains an accurate internal disk simulator with support for various layouts of data on multiple disks. The applications are taken from a suite collected by Patterson, Gibson *et al.* from various application domains (databases, speech recognition, out-of-core scientific computing, etc) and are used in their evaluation of the TIP2 system. They modified each application to provide hints about its upcoming accesses wherever possible. The trace collection system[1] captures these hints as they are presented to the operating system, allowing the simulator to model both hinted and unhinted accesses, and situations in which hints "trickle in" over time. Earlier comparisons performed by this group and others studied the model in which all accesses are hinted, and all hints are available at the start of the simulation — in the TIP2 application suite both of these assumptions are far from true, so I am fortunate to have the new traces available.

Chapter 4 describes the four algorithms I compare. The first is the TIP2 algorithm of Patterson *et al.*, mentioned above. Second is another existing system from the literature: LRU-SP/AGGRESSIVE, by Cao *et al.* [CFKL95b, CFL94a]. These two systems take different approaches to prefetching within a single stream of hints. Tracy Kimbrel and myself with a number of collaborators compared the two internal prefetching algorithms on single hinted streams [KTP+96], and found that a hybrid of the two approaches called FORESTALL performed better than either. Therefore, I also consider each of these two systems augmented to include FORESTALL-based prefetching. The extension of LRU-SP/AGGRESSIVE is straightforward and results in LRU-SP/FORESTALL. The extension of TIP2 is nontrivial, and results in a new algorithm called TIPTOE, or TIP with Temporal Overload Estimators. I describe TIPTOE and LRU-SP/FORESTALL, and their predecessors TIP2 and LRU-SP/AGGRESSIVE.

In Chapter 5, I describe a study performed with Hugo Patterson and Garth Gibson at CMU, in collaboration with another group of researchers: Tracy Kimbrel, Anna Karlin and Brian Bershad at the University of Washington; Pei Cao at the University of Wisconsin; and Kai Li and Ed Felten at Princeton. This study focuses on the case of a single process prefetching and caching its data; the results appear in [KTP+96]. We draw several conclusions. To summarize, TIP2 will sometimes allow the disk to idle in situations that would benefit from more aggressive prefetching. LRU-SP/AGGRESSIVE on the other hand can sometimes prefetch *too* aggressively. TIPTOE and LRU-SP/FORESTALL, which use the hybrid algorithm FORESTALL, combine the benefits of FIXED-HORIZON and AGGRESSIVE by dynamically estimating upcoming load based on whatever hints are present and deciding how aggressively to prefetch.

Finally, in Chapter 6 I describe a study performed with Hugo Patterson and Garth Gibson [TPG97] that considers prefetching and caching in a multiprogramming environment. We find that COST-BENEFIT, as implemented in TIP2 and TIPTOE, typically finds

[1] I am indebted to David Rochberg for creating, post-processing and providing these traces.

better resource allocations than does LRU-SP, especially in situations in which process consumption rate does not match re-use (for instance, COST-BENEFIT would tend to perform well if a fast process with little re-use were to run beside a slow process with significant re-use). Additionally, TIPTOE performs better than TIP2 when hinted data needs to be prefetched far in advance, or cached for distant re-use.

## 1.2 Part II: Weighted Caching and Free Time

The theoretical contribution of the thesis is given in Part II; the results described are joint with Avrim Blum and Merrick Furst, and benefit from later collaborations with Carl Burch and Yair Bartal. Again, I present a quick "big picture" description, and then a roadmap and summary of the theory part of the document.

### 1.2.1 The Big Picture

*Online algorithms* process a sequence of requests, making decisions about early requests before seeing later requests. Informally, we face on-line problems routinely. For instance, when driving on the highway, we must choose a lane. At each point in time we may stay in our current lane or, with some effort, switch lanes. The central problem here is the same as the central problem in the formal version of the problem; namely, if we switch lanes because the other lane is moving faster, our new lane will immediately slow down and we will have made the wrong decision because we didn't know the future. Consider the following specific online problem:

**The cache-management problem:** Given a main memory of $n$ elements and a cache of $k$ elements, service a sequence of requests in an online manner. Each cache element may hold any element of main memory. Requests for memory elements that reside in the cache incur no cost. Requests for elements that are not in cache must be loaded into the cache, evicting some cached element to make room. The cost of servicing a sequence of requests is the number of accesses to main memory.

This problem is an abstraction of a typical caching problem, phrased formally to allow a precise theoretical evaluation of approaches. Other caching problems can be phrased similarly; for instance, we could phrase the same problem at a different level of the storage hierarchy, caching disk blocks in main memory. We could then add the constraint that some disk blocks lie on "local" disks with low latency, while other blocks lie on more distant servers with higher latency. The resulting variant, called *weighted caching*, is the central problem of Part II:

**The weighted caching problem:** Given $n$ disk blocks, each with an associated positive *weight*, and a memory capable of storing $k$ disk blocks, service a sequence of requests in an online manner. A request for a disk block that resides in memory incurs no cost. A request for a block that is not in memory must be fetched from disk, evicting some cached element to make room, at cost equal to the weight of the block. The cost of servicing a sequence of requests is the sum of the costs of servicing each request.

Weighted caching has been studied in its own right since online algorithms were first introduced in [MMS88a]. We came to the problem, however, because it was central to an extension we were considering to the traditional online model. In the standard model, an algorithm is asked to process a request sequence. Each request is presented after the algorithm has completed processing the previous one and the cost of the algorithm is the cumulative time or work needed. In many natural on-line settings, however, requests may arrive infrequently relative to the speed of the algorithm. In these situations, it makes sense to model an algorithm as having *free time*, for which it is not charged, between the servicing of one request and the arrival of the next. For instance, a classical example on-line problem is the "servers are fire trucks" problem in which requests represent fires, and when a request arrives some server, or fire truck, must be moved to the fire as quickly as possible. In this example, one rightly cares much more about the time it takes to get a fire truck to a fire once a call has been made and cares much less about any time spent moving trucks to resting places while there are no fires to attend to. In general, in computing situations, if the process issuing requests is substantially slower than the process serving requests then the server is liable to have a fair amount of free time at its disposal between demands. In these situations it makes sense for the serving algorithm to use the free time between requests to position itself advantageously, rather than idly waiting for the next thing to do. Our results for the weighted caching problem can be extended to give general algorithms in the free-time model.

### 1.2.2 Theory Roadmap

Chapter 7 gives a more complete introduction to the field of online algorithms, a summary of the results to follow, and a discussion of related work. I give a quick introduction here that contains a subset of the information in Chapter 7. Section 10.3 of the conclusion then discusses this work as it relates to the systems work introduced above. The remainder of this section assumes familiarity with $k$-server problems and the competitive ratio. If these terms are not familiar, the presentation in Chapter 7 includes all the necessary definitions.

Chapter 8 considers weighted caching. Phrased in the language of $k$-servers, a weighted caching problem is a $k$-server problem in which the cost to move a server from one point to another is equal to the *weight* of the destination point, regardless of the source: $d_{ij} = w_j$.[2]

[2]Note that spaces with $d_{ij} = w_i$, or the symmetric version with $d_{ij} = 1/2(w_i + w_j)$, are all within a small additive constant of the definition as given.

As an example, consider a web browser that may store some fixed number of pages (for instance, assume that all the pages are text-only, and so are roughly the same size). The "weight" of a page is the time it takes to read the page from its web server. The task of determining which page should be evicted when a new page is loaded is exactly a weighted caching problem. We give an $O(\log^2 k)$-competitive randomized algorithm for $(k+1)$-point spaces, and an $\Omega(\log k)$ lower bound on the competitive ratio of any online algorithm for any fixed weighted-cache space of $k+1$ points or more. That is, the lower bound holds for any space, not just for a single constructed space.

Next, Chapter 9 discusses free time. The precise model we consider is the following: whenever a request arrives, the server algorithm must service it and the charge is the standard notion of cost. However, once the request is serviced, the server algorithm may adjust its configuration as desired without charge. The cost of running a server algorithm with free time is compared with the cost of running the optimal off-line server algorithm *without* free time.[3] At first glance this comparison might seem unfair, but in fact we show in Section 9.3 that for deterministic algorithms free time helps by at most a small constant factor. We give an $O(\log^2 k)$-competitive algorithm for general $(k+1)$-point spaces in the free-time model by reducing the problem to a standard server problem on a weighted cache space. We also show that the $\Omega(\log k)$ lower bound for arbitrary weighted cache spaces mentioned above generalizes to algorithms with free time.

We then show a number of related results, exploring the model. Unlike the standard on-line model for which there exists a general $\Omega(\sqrt{\log k/\log\log k})$ lower bound for any space [BKRS92], Section 9.4 shows that in the free-time model there exist metric spaces with constant competitive ratio. Section 9.5 then considers algorithms that are given possibly erroneous hints to guide their activity during free time. Next, Section 9.6 considers bounded rather than infinite free time for certain metric spaces. And finally, Section 9.7 presents an algorithm that a computer might use to pre-process potential future commands while waiting for a user to type, taking into account that different commands will have different durations and so different potential savings.

As a final note, the results for weighted caching are no longer the most general known. Since this work was done, results of Bartal, Blum, Burch and myself [BBBT96] give an $O(\log^6 k)$-competitive algorithm for arbitrary $(k+1)$-point spaces that also gives polylog competitive ratio for $k + \text{polylog}(k)$-point spaces.

[3] For server problems, the optimal off-line cost *with* free time is 0.

# Part I

# Systems

# Chapter 2

# Systems Introduction

*That's a fine idea in practice, but it will never work in theory.*

*— old French joke*

Part I of the thesis motivates and answers the following question: If applications provide information to the operating system about future I/O accesses, how can this information be used, and what improvement can be attained? This chapter motivates the work in some detail, presents an overview of my contributions, and discusses related work.

## 2.1 Motivation

As described in Section 1.1.1, the most dramatic benefits of informed prefetching and caching arise in the presence of storage parallelism, as provided by arrays of disks. However, most popular general-purpose operating systems are not designed to support parallel I/O. Primitives tend to be synchronous to provide programmers with the useful invariant that executing an I/O instruction guarantees that the data will be available when the call completes. However under this abstraction, each process will have at most one outstanding I/O.[1] Since our goal is to keep many disks of an array busy simultaneously this paradigm must be extended; a natural approach is to provide information to the operating system about multiple future I/Os to expose the parallelism in the request sequence.[2]

[1]Alternatively, processes can access disk parallelism through asynchronous I/O. Patterson, Gibson *et al.* argue that, for the programmer, providing hints is both easier and more effective than providing parallelism through AIO.

[2]These techniques are also effective for single-disk systems. Most importantly, hints about future I/O's allow the system to make better caching decisions, reducing the number of I/O's that are necessary.

There have been two classes of API's proposed to allow the application to provide information to the operating system about its upcoming accesses. The first is cache management advice of the form, "Cache file FOO.DATA using the MRU cache replacement scheme." This approach has been implemented by Cao *et al.* [Cao96, CFL94b]. The second approach, known as *disclosure*, has been taken by Patterson *et al.*, and also by Cao *et al.*. A disclosure is a hint about future accesses given in the language of the existing I/O interface. Thus, in Unix a disclosure of future accesses might be a list of file descriptor and byte range pairs.[3] Patterson *et al.* suggest that disclosure has three advantages over cache management advice. First, it is a more portable interface; the correct cache management algorithm might be machine dependent, so code would have to be re-written under an advice policy. Second, if the system is unable to honor the optimal policy because of resource limitations, disclosure provides the system with information to evaluate the effectiveness of other (possibly system-dependent) options. Third, disclosure adheres more closely to software engineering principles of modularity because it allows hints to be specified at the same level and in the same language as the I/O calls themselves.

We have noticed two additional advantages of disclosure. First, if random accesses to a file are disclosed, there may be opportunity for substantial re-use if the file is not too much larger than the cache. Our experiments include two database joins; in one of these joins there are enough hits to an indexed inner relation to provide substantial re-use of the inner relation data blocks. Under a disclosure policy, more data can be cached than under random evictions. Second, some applications have access and re-use patterns within a file that do not fit traditional advice schemes — the file is not read sequentially, cyclically, strided, parallel-strided, randomly, or according to any other simple plan. Advice schemes would classify such an access pattern as random and would expect no caching benefit, but there might be substantial re-use possible. We experiment with a scientific visualization program that takes 2-D slices through a 3-D dataset at arbitrary angles. The accesses are not random and, depending on the size of the dataset, there may be substantial re-use. Disclosure will provide this re-use when possible, based on the size of the cache; random replacement will do so much less effectively. For these reasons, I have adopted disclosure as the form of communication between the application and the system about upcoming I/O accesses.

But how difficult is it to generate these disclosures? For code that is being written with such a system in mind, Patterson *et al.* argue that it is both easier and more effective to

Second, hints allow the operating system to reduce average I/O time by performing more effective disk scheduling. Third, hints allow the system to overlap I/O and computation — even if the application is bursty and would ordinarily compute without any I/O for a substantial interval of time. Nonetheless, the potential advantages grow as the array size increases.

[3] By providing lists of multiple accesses, system call overhead can be amortized over many I/O requests.

disclose future accesses than to implement application prefetching by asynchronous I/O. Chapter 3 describes modifications made by Patterson *et al.* to a suite of I/O-intensive applications to support disclosure. These modifications suggest that it is possible to provide disclosures in a range of situations, in a relatively straightforward manner — see Section 3.2 for details. If the software has already been written but the source code is available, recent work by Mowry, Demke and Kreiger has shown that compilers can induce some programs to disclose their future accesses automatically [MDK96], especially in the realm of scientific computing. Section 2.3 describes these results in more detail. Finally, although no research has yet been published about this approach, speculative execution offers the opportunity to discover future reads that are not too strongly dependent on data that has not yet arrived. When a program stalls for I/O and the CPU would otherwise be idle, the system may continue to execute the program speculatively in a "sandbox" [WLAG93] so as to discover future accesses without corrupting existing state. This approach offers the additional advantage that source code is not required. Thus, there is evidence to suggest that compilers or programmers can provide disclosures, and there are preliminary approaches to generating disclosures transparently.

Patterson *et al.* have presented a fully operational system to perform informed prefetching and caching using an algorithm called TIP2, described in detail in their paper and in Chapter 4 of this document. Figure 1.1 shows the reduction in stall provided by TIP2 relative to OSF1 for a variety of disk array sizes, averaged over a suite of I/O-intensive benchmarks. The figure demonstrates that knowledge of future accesses can result in dramatic improvements, especially in the presence of storage parallelism. Furthermore, Digital's OSF1 operating system already supports an aggressive readahead strategy that will submit up to eight clusters of eight blocks (each block is eight KByte) to the driver at once. For sequential reads OSF1 will be able to keep many disks of the disk array busy; the fact that TIP2 performs substantially better suggests that sequential readahead schemes, even extremely aggressive schemes like OSF1's, are insufficient to reliably keep the disks busy fetching needed data.

Another system presented by Pei Cao and collaborators [Cao96] has shown a reduction of up to 50% in overall execution time on a different suite of applications, also modified to give hints to an operating system designed to accept them. In this case, the system had a single disk and the improvement was due largely to cache management. Thus, independent groups have shown that disclosures are capable of conferring dramatic reductions in I/O stall time.

Before the work described in this document, all results in this area followed the general format of Figure 1.1: a system for informed prefetching and caching was shown to provide significant improvements over a system without this functionality. However, given that application disclosures are capable of reducing I/O stall time by 83%, it is clearly of interest to understand exactly how well various approaches to this problem perform relative to one another, and to identify or create the best possible approach. This part of the thesis makes two contributions: the first is to provide a detailed comparison of approaches to the problem, and the second is to use the results of this comparison to generate the best known algorithm for the problem.

## 2.2 Overview

In order to understand how well a particular system is using a sequence of disclosures, I break the problem into two distinct pieces and study each separately. First, a system must manage prefetching and caching within a single stream of disclosed accesses, given a fixed set of resources. That is, if a single process runs alone on a machine and discloses all of its accesses, which data should be prefetched at which point, and which resident blocks should be evicted? Notationally, I will refer to this piece of the overall problem as the *Standalone Prefetching And Cache Eviction* problem, or the *SPACE* problem for short. Second, a system must address the *allocation problem* of dividing disk and cache resources among multiple hinted and unhinted access streams. In a complete system, the allocation algorithm partitions resources among competing processes, and the SPACE algorithm manages prefetching within each stream, as shown in Figure 2.1.

Chapter 5 addresses the SPACE problem: prefetching and caching within a single stream. These results were derived in a recent collaborative study with Kimbrel, Patterson, Cao, Gibson, Karlin, Bershad, Felten, and Li [KTP+96], which analyzes prefetching and caching algorithms in the context of a single process disclosing all its accesses at startup. In this domain, all resources are dedicated to the single hinted stream, so the allocation problem does not arise, and only the SPACE problem need be solved.

We consider two existing algorithms, and based on this study, suggest a new algorithm combining the advantages of the other two. First, the TIP2 system addresses both the SPACE problem and the allocation problem — we extract the SPACE algorithm and use it alone. TIP2 was designed using a system model that assumes essentially infinite storage parallelism, as would be delivered by a large disk array. Restricted to the single-process fully-hinted domain, the TIP2 system model results in a conservative prefetching algorithm that prefetches only those blocks that will be read within a small, fixed number of accesses. This conservative approach may fail to prefetch deeply enough into the request stream when disk parallelism is limited or when the I/O workload is highly unbalanced across the disks of an array. The second algorithm we consider, the AGGRESSIVE algorithm of Cao *et al.* [CFKL95b], prefetches deeply without regard to disk load. AGGRESSIVE will not allow the disk to go idle when there is missing data and reasonable eviction decisions exist, but it may incur substantial computational overhead by under-valuing caching and performing many unnecessary I/O's when there is ample

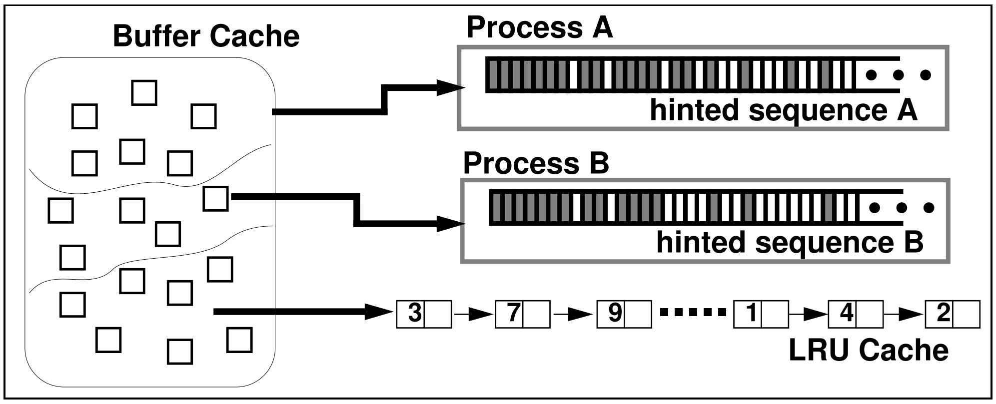



Figure 2.1: The Multi-Process Informed Prefetching and Caching Problem. A system to solve this problem must solve two independent subproblems. First, it must solve the *allocation problem* by partitioning resources among multiple competing processes, each of which may be disclosing arbitrary fractions of its accesses. Second, it must solve the *SPACE problem* (standalone prefetching and cache eviction), deciding when each process should prefetch, and what it should evict given the constraints imposed by the allocation algorithm.

disk bandwidth. To resolve the tension between over-conservative and over-aggressive prefetching our collaboration developed a new algorithm, FORESTALL, whose prefetching behavior is based on a dynamic estimate of upcoming disk load. When future disk load is high, FORESTALL prefetches more aggressively, but when load is low FORESTALL does not fetch far ahead into the request stream. This algorithm performs well in the single-process fully-hinted domain on a variety of trace-driven simulation comparisons to the load-oblivious approaches mentioned above. The results shown in Chapter 5 confirm the results of our collaborative study, but are performed in a new environment using a new set of traces that capture additional effects. These traces are introduced below, and are described in detail in Chapter 3. [4]

Chapter 6 addresses the allocation problem: how should resources be allocated among multiple, competing processes that may disclose arbitrary fractions of their accesses. Al-

[4]Before developing FORESTALL, our collaboration also considered a fourth algorithm called REVERSE-AGGRESSIVE, due to Kimbrel and Karlin, which is guaranteed to be within a small constant factor of optimal on every sequence for a particular system model [KK96a]. This is the only algorithm for which such a guarantee is known. We found that REVERSE-AGGRESSIVE performs well in all situations, but is difficult and expensive to implement, while FORESTALL performs as well as REVERSE-AGGRESSIVE and is both simpler and cheaper to implement. For this reason, Chapter 5 focuses on FORESTALL and does not study REVERSE-AGGRESSIVE. A brief description of REVERSE-AGGRESSIVE appears in Section 4.2.2.

location algorithms can only be evaluated in tandem with SPACE algorithms. I consider two existing allocation algorithms, each paired with the SPACE algorithm with which it was initially presented, and then extend each allocation algorithm to incorporate adaptive prefetching (as provided by FORESTALL) according to the lessons learned in the single-process case. First, the TIP2 system makes resource allocation decisions by weighing the benefit of providing resources to a consumer against the cost of taking them from a supplier. This *cost-benefit* allocation framework provides a general, extensible method for reasoning about allocation decisions. As discussed above, TIP2's cost-benefit allocation algorithm was initially presented in tandem with a conservative fixed-depth prefetching scheme. I show how to extend cost-benefit allocation to incorporate FORESTALL's adaptive single-process prefetching algorithm, resulting in a new algorithm called TIPTOE, or TIP with Temporal Overload Estimators.

The next system I consider is LRU-SP, due to Cao *et al.* [CFL94a, Cao96], an extension of traditional LRU replacement. In this system, buffers in the file cache are tagged with an "owner."[5] When the kernel must evict a block it chooses the process that owns the global LRU block; that process must then decide, possibly based on application-specific knowledge, which of its blocks to give up. In the original system individual processes used the AGGRESSIVE algorithm to prefetch, resulting in LRU-SP/AGGRESSIVE. It is straightforward to implement the FORESTALL algorithm with LRU-SP, resulting in LRU-SP/FORESTALL. With TIPTOE and LRU-SP/FORESTALL we have comparable prefetching and caching components and dramatically different allocation strategies. Chapter 6 analyzes these two informed prefetching and caching systems, contrasting them to each other and to their predecessor systems, TIP2 and LRU-SP/AGGRESSIVE.

Finally, I give a quick overview of the trace-driven simulation system used to perform these experiments. As mentioned above, Patterson *et al.* modified a suite of six applications from various application domains (databases, speech recognition, out-of-core scientific computing, etc) to disclose their accesses to the system. I use traces of the modified versions of these applications.[6] The tracing tools capture and timestamp both hints and accesses, allowing accurate modeling of the implications of late-arriving hints and unhinted accesses. They also capture context switches, allowing an accurate measurement of inter-access process compute time. I developed a disk-accurate, trace-driven simulator based on the RaidSim simulator [Lee89], which generated the results given in Chapters 5 and 6. Details about the simulator, the applications, and the traces are given in Chapter 3.

I describe three primary results. First, in the single-process domain FORESTALL outperforms both conservative and aggressive non-adaptive techniques. Second, in the multi-process domain COST-BENEFIT outperforms LRU-SP. And third, TIPTOE combines

[5] The owner of a block is the last process to access that block.

[6] I am indebted to David Rochberg for creating, post-processing and providing these traces.

these two results into a single algorithm that performs well across the board, improving execution time on average across a series of hinted two-process experiments by 12% relative to LRU-SP/AGGRESSIVE, 10% relative to LRU-SP/FORESTALL and 3% relative to TIP2.

## 2.3 Related Work

The work in this dissertation is a direct extension of work by Hugo Patterson, Garth Gibson and their collaborators on the TIP2 system [PGG+95]. My work is also closely related, and in some cases collaborative with, work of Pei Cao, Tracy Kimbrel, Anna Karlin and others on LRU-SP, AGGRESSIVE and related algorithms [KTP+96, Cao96, CFKL95b, CFKL95a, CFL94a, CFL94b]. Upcoming chapters describe these systems in detail, so I will not include a discussion of them here.

At a high level, the work in this thesis represents an approach to providing and using more detailed access information in the I/O subsystem. There have been many other approaches to this problem, centered on three topics: non-traditional cache management, prefetching, and hints. I will discuss work from other subfields that touch on one or more of these questions.

### 2.3.1 Prefetching from Disk

In this section I discuss prefetching from disk under both informed and predictive models. The distinction does not depend on the correctness of the hints, but on the nature of the hints. An *informed prefetching* technique is given the blocks or files and must decide when and if to prefetch. A *predictive prefetching* technique includes a component to determine the blocks or files that should be prefetched.

#### Informed Prefetching

This thesis is concerned with informed prefetching, in which the application discloses the data it will require. Though most related work addresses either cache management or predictive prefetching, there are a few other non-predictive approaches to prefetching.

Steere and Satyanarayanan [SS95] present a general abstraction called *dynamic sets* to allow parallel prefetching of objects that will be required in the near future. It is possible, for instance, to load a web page then specify a dynamic set containing all links on the page. The server could then prefetch the links in parallel.

Grimshaw and Loyot [GJ91] present ELFS, an extensible object-oriented file system with build-in support for prefetching and cache management. Objects can specify appropriate prefetching and caching behaviors based on specific knowledge about access patterns and re-use. For instance, the default system includes a type of "file" called a "2D_matrix_file." This type of file includes a blocked layout for quick scanning in both row and column-major orders, and a prefetch policy to generate the next row or column as the first one is being consumed. Thus, policy decisions are tied to the objects that require them, and can be specified at the level of detail appropriate for the task.

A number of researchers in mobile computing have considered the problem of prefetching data to allow disconnected operation. Kistler and Satyanarayanan of the CMU Coda project [KS92] are concerned with providing an environment for a completely disconnected user. They take the informed approach that user-level tools can allow an appropriate cache state to be specified by hand, and this data can then be fetched or kept resident when disconnected operation is expected.

Finally, Mowry, Demke and Krieger [MDK96] present a compiler to produce executables that automatically generate prefetch requests for data that will be accessed in the near future. The compiler is designed for scientific codes, and performs static analysis to uncover many common access patterns, including complex strided accesses. A runtime component tracks cache state to filter out prefetches that are already in core before a system call is generated. The technique successfully hid more than half the I/O latency is seven of eight NAS parallel benchmarks.

### Predictive Disk Prefetching

The most common incarnation of predictive prefetching is sequential readahead, ranging from one-block lookahead as described by Smith [Smi85] and implemented in Multics [FO71] and Unix [MJLF84] to more aggressive versions such as Digital's OSF 1 operating system, which will prefetch up to sixty-four 8KByte blocks in advance. Similarly, there have been efforts to provide higher throughput by reading larger objects, as in [MK91].

But recently several researchers have addressed predictive prefetching more speculatively, prefetching entire files before they are referenced, or predicting accesses across files. The primary advantage of this type of predictive prefetching versus informed prefetching is that applications do not need to be modified: the system makes predictions about future accesses (usually based on past accesses) and prefetches accordingly. Griffioen and Appleton have presented a series of results of this flavor. Initially [GA93] they considered wide-area file systems, in which latencies are extremely high and prefetching at the file level affords the opportunity for large savings. They later applied the same approach to general high-latency storage [GA94, GA95]. Prediction of future file accesses is performed by a data structure called the *probability graph*, in which files are nodes, and edges represent sequential (or nearly sequential) accesses or invocations. Thus, if program A is read, and either calls or reads program B soon after, there will be an edge between A and B. The edges are weighted according to the probability of the sequential call, over a large trace. The graph is used to predict future file accesses, and the predictions are then used to initiate prefetching. They showed that cache miss rates can be reduced by up to 40% for some workloads. In [GA96] the same authors give an implementation of the system under SunOS.

In the domain of predictive prefetching for mobile computing, Kuenning *et al.* [KPR94] analyzed trace data and suggested that the necessary cache contents could be generated automatically using predictive approaches. Tait and Duchamp [TD91] give a predictive mechanism called the *working graph*; they show that predictive prefetching using this structure performs better than LRU and has small overhead, for the domain of low-bandwidth connected mobile computing. Later work by Lei and Duchamp [LD97] gives an implementation of such a prefetching scheme, here based on *access trees*. Their results show a reduction in cache miss rate of up to 47%, and a reduction in application latency of up to 40%.

Korner [Kor90] presents a predictive technique based on file extenders that is used primarily to perform informed cache management, although the work also includes a prefetching component. An example rule used in the cache manager might be, if a file ends with ".out" then don't cache the file. In simulations the approach is shown to improve performance by up to 340%.

There have also been approaches to predictive prefetching in the database community. Palmer and Zdonik [PZ91] present Fido, a cache manager that employs an associative memory technique to predict future accesses in a particular isolated context. They showed some early simulation results suggesting that predictive prefetching within the database domain is a promising approach.

More recently, Curewitz, Krishnan and Vitter [CKV93] present a theoretically-grounded approach to predictive prefetching based on the observation that a succinct representation of a sequence of requests must capture some internal structure within the sequence. The representation should therefore be useful for predicting future elements of the sequence. They formalize this notion of "prefetching via data compression," and give predictors that perform well compared to Fido's associative memory approach.

### 2.3.2 Database Cache Management

Stonebraker [Sto81] points out that databases often have access to detailed information about access patterns, relative to general-purpose file systems, and should be able to incorporate this information into resource management decisions. Selinger *et al.* [SAC$^+$79], for instance, use access knowledge within the DBMS, employing a simple model of page re-use to estimate the cost of various alternative access paths and join orders, taking into account both I/O and CPU costs. Stonebraker goes on to argue that some form of "advice" should be passed from the database management system to the cache manager. Chou *et al.* [CDKK85] give a general interface that allows application programs to mark individual pages with a priority representing the application's view of the importance of the page.

Within a database buffer manager, which has more complete information about the types of queries being performed, the most successful early approach to management is the *hot set* buffer pool management model of Sacco and Schkolnick [SS82], an extension of the working set model of Denning [Den68]. The hot set model provides estimates of the advantage a query will derive from a certain amount of buffer cache resources, which can then be used to drive allocation decisions.[7] However the original hot set model focuses on allocation of buffers among processes, rather than cache management of buffers within a process. The *query locality set model* (QLSM) of Chou and Dewitt [CD85] also specifies eviction policies within a particular process.[8] For instance, if data is being read cyclically then MRU is the most appropriate policy. Around the same time, the original hot set model was extended by its proponents to include different access patterns, so that all operations supported by IBM's System R database test-bed could be represented in the model [SS86].

Cornell and Yu [CY89] integrate query optimization with buffer cache management. They phrase the problem as a 0-1 integer linear program, which they approximate to choose query plans based on buffer restrictions. Later, in [YC91], the same authors present a different global minimization technique based on simulated annealing for the same problem.

These approaches use the nature of the query to make allocation decisions before the query actually executes — I will call such information *static* because it does not depend on the execution of the query. Alternatively, traditional buffer replacement under LRU is entirely *dynamic* in the sense that decisions are made solely on the basis of the buffer cache state, and are performed while the query executes. There have been other purely dynamic approaches. O'Neil *et al.* [OOW93] extend the LRU algorithm to LRU-$k$, a

[7] The earlier working set model makes similar estimates based on the number of buffers a process touched, without reference to data re-use — thus processes that moved quickly through data, without re-use, would be given inappropriately large allocations. The same effect arises in general file system cache management, and is in fact on the most significant effects differentiating the algorithms studied in this thesis. O'Neil *et al.* [OOW93] describe another approach to this problem, discussed below.

[8] The same general line of research led to the work in this thesis. Work on TIP2 [PGG+95] addressed both parts of the problem but was more strongly focused on allocation of resources among multiple processes. Once the system was built, more recent work [KTP+96, TPG97] focused more specifically on cache management within a process, and on integrating the two problems.

dynamic algorithm that takes re-use into account. In traditional LRU, which corresponds to LRU-1 in their model, the time since the most recent reference to a page is taken as an estimate of the value of the page. In LRU-2, the time since the second most recent reference is used as the estimate of value. Thus, a page that has been accessed once but is not re-used is given small value.

Another line of research combines dynamic and static information. Most closely related to the work described in this thesis is the *marginal gains* approach of Ng, Faloutsos and Sellis [NFS91] (in fact, their approach inspired the cost-benefit approach of TIP2). They define the marginal gain of using $s$ buffers to service a particular reference as the difference between the expected number of faults with $s$ buffers versus $s-1$ buffers. For instance, in a looping reference over $N$ items, adding one buffer would decrease the number of cache misses by approximately 1 every $N$ accesses. Using these estimates, they propose an algorithm MG-$x$-$y$ that makes allocation decisions based on runtime buffer availability. In [FNS91] the same authors generalized the conditions in which a query may be admitted for processing. In all these approaches, however, queries that are not sequential or looping are considered to be random, and their estimates are created accordingly. Chen and Roussopoulos [CR93] extend the marginal gains of other queries based on a sampling of their re-use under the LRU policy. This idea also contributed to the development of TIP2.

### 2.3.3 I/O for MIMD Multiprocessors

Work of Kotz and his collaborators focuses on prefetching for MIMD multiprocessors to hide the latency of high-performance I/O subsystems. In the domain of high-performance coarse-grain parallel computering several factors make prefetching more difficult. Often only one thread executes on a single node, so multiprogramming cannot hide the latency of a demand fetch. The I/O loads are also unusual in that, while there is high sequentiality within each node, multiple streams of sequential requests may be interleaved; worse yet, optimizing the cache hit ratio may not optimize the completion time, because speeding up the first node to reach a synchronization barrier may not improve performance. Kotz and Ellis [KE90] consider prefetching with perfect knowledge of future accesses, and show that the technique often improves the cache hit ratio *and* reduces overall execution time. However in some situations prefetching actually hurts performance. In [KE91, KE93] the same authors extend their results to prefetch predictively rather than assuming *a priori* knowledge of future requests. They develop simple recognizers of traditional access patterns and develop prefetching policies that are specific to the patterns. They report that, in general, prefetching is beneficial in their setting.

Kotz [Kot94] goes on to argue for *Disk-Directed I/O*, an informed prefetching technique in which high-level structure is passed to the I/O subsystems. For instance, a multi-threaded application can inform the system that a series of interleaved sequential reads is beginning.

### 2.3.4 Prefetching and Virtual Memory

Both prefetching and informed cache management have been incorporated into virtual memory systems. Trivedi [Tri79] studied prefetching of virtual memory pages, referred to as *prepaging*. He argued that predictive techniques are not sufficient to prefetch data across *phase transition* boundaries, in which the behavior of the user changes dramatically (*e.g.*, when a user switches from word processing to compilation). Based on this conclusion, he advocates *prepaging advice* given by the programmer or the compiler to the system. Several researchers have incorporated user-level paging managers into existing operating systems. McNamee and Armstrong [MA90] added user-level paging control to Mach; Harty and Cheriton [HC92] extended the V++ kernel in the same way. Finally, Krueger *et al.* [KLVA93] present a general toolkit and set of profiling tools to provide user-level control of paging.

### 2.3.5 Prefetching From Main Memory

There is a large body of work on prefetching *from* main memory, rather than *to* main memory. Smith [Smi78] showed that large-block prefetching to cache may be ineffective, and while prefetching is potentially effective, small details of the implementation can have a significant effect on the overall performance. Baer and Chen [BC91] describe a hardware scheme for pre-loading a cache in the presence of sequential accesses to memory. Rogers and Li [RL92] describe a simple hardware modification and compiler support to allow speculative loading of data into cache, which results in significant reductions in average memory latency for a series of benchmarks. Chen and Baer [CB92] suggest a combination of hardware-based cache prefetching and hardware support for non-blocking caches. Tullsen and Eggers [TE93] showed that some architectures (in this case, a high-latency bus-based multiprocessor) are not well-suited for prefetching. Mowry, Lam and Gupta [MLG92], on the other hand, showed that careful software-controlled compiler-directed prefetching for scientific codes provided substantial improvements in performance, reducing execution time of some benchmarks by a factor of 2. To summarize, the community has not converged on a single effective solution to this problem, and while the advantages can be substantial, the margins are tight – the more complex reasoning about prefetches that is routine in disk prefetching work is not feasible in this domain.

### 2.3.6 Theoretical Treatments

The first theoretical treatment of caching was a study by Belady in 1966 [Bel66], which introduced the problem and studied many algorithms, including the MIN algorithm, which makes optimal replacement decisions when future accesses are known. Aho, Denning and Ullman [ADU71], and later Coffman and Denning [EGCD73], study page replacement under particular probabilistic models of future page arrivals. In the presence of variable-sized buffers there have been a number of extensions to this work, beginning with Prieve and Fabry's VMIN algorithm [PF76], and more recently, work to derive practical versions of VMIN, such as Choi and Ruschitzka's work on SETVMIN [CR96].

In the case of multiple processes that give hints and compete for cache resources, Barve, Grove and Vitter [BGV95] describe a competitive approach to cache management. If the hints are perfect and only the interleaving is unknown, they show an algorithm with a competitive factor logarithmic in the number of processes. This means that, even if the relative rates of the processes vary in an adversarial manner, the algorithm is guaranteed to be within the given factor of optimal in terms of number of cache misses.

# Chapter 3

# Simulation Environment

*So very difficult a matter is it to trace and find out the truth of anything by history.*

*— Plutarch, "Plutarch's Lives"*

All the experiments described in later chapters are performed using trace-driven simulation. Section 3.1 motivates this decision and describes which effects are captured by the tracing and simulation environment and which are not.

As described in Section 2.1, Patterson, Gibson *et al.* have developed a suite of I/O-intensive applications to benchmark implemented systems for informed prefetching and caching. Section 3.2 describes this suite. I make use of a set of detailed traces of these applications running on modern equipment with representative inputs; the trace-collection environment is described in Section 3.3 and the individual traces are discussed in Section 3.4. All of the applications have been tailored to provide hints to the existing TIP kernel about upcoming accesses. The traces capture these application-provided hints as they are delivered to model the effects of unhinted requests and late-arriving hints. The traces also capture process compute time between successive I/O requests to allow accurate modeling of variable periods of inter-access computation. Finally, the simulator itself and the associated disk simulator are described in Section 3.5.

## 3.1 Why Trace-Driven Simulation?

The primary advantage to simulating the algorithms versus implementing them in a real system is control. Because the environment is more tightly controlled and more manageable than a full kernel, the algorithms can be developed, debugged and profiled more quickly and effectively. In the same amount of time the implementer can code more algorithms, perform more experiments, and understand the results more deeply.

Additionally, a simulator allows control over parts of the system that are typically fixed or difficult to change for a particular execution environment. For instance, it is simple in a simulated environment to modify the size of the on-disk cache, the clock speed of the processor, the details of the disk queueing discipline, the overhead associated with submitting an I/O, and so on. This allows us, for example, to examine the same algorithms in "future" environments estimated by extrapolating current technology trends.

Similarly, it is possible to examine different approaches to giving hints within an application simply by re-processing the trace file, without modifying the application itself.

The downside, of course, is that an actual implementation may uncover factors that did not appear in simulation. The simulator is described in detail in Section 3.5, but I give a high-level description here to summarize which effects are captured and which are not.

### 3.1.1 Disks and Disk Drivers

The simulator implements a number of disk scheduling disciplines. Most experiments are performed using CSCAN scheduling, following [SCO90]. The disk simulator, described in Section 3.6, simulates the HP97560 disk drive [RW94, KTR94]; it models head position, and computes seek, rotate and transfer latencies in the usual manner. It also includes a non-segmented on-disk cache, and provides a simple model of SCSI bus overhead. The simulator has been validated against the simulator of Kotz *et al.* [KTR94]. Since the traces capture disk blocks, the simulator correctly models file layout on the disk.

### 3.1.2 Processor Scheduling

The simulator has an embedded round-robin CPU scheduling algorithm. However, since our applications are strongly I/O bound they rarely compute sequentially for long enough to cause a context switch; almost all context switching occurs when processes block for I/O. This means that the choice of scheduling algorithm is not critical. I do not consider prioritized schemes; if the environment were to be applied to other workloads this decision might have to be revisited.

### 3.1.3 Buffer Cache

The simulator includes a buffer cache manager and a set of modules implementing different replacement policies. For the traces and the simulator, disk blocks, file system blocks and fragments are all set to 8 KByte. The buffer cache manager does not implement the code (mainly fault handling) to allocate multiple buffers to a single disk block read.

### 3.1.4 Inaccuracies in the Simulation Environment

The simulation environment does not model virtual memory: the trace-collection mechanism does not capture virtual memory paging, and the simulator does not include a virtual memory system. Incorporating VM would require extending the trace-collection tools and the simulator to operate at the level of memory references, which would entail substantial modifications to both toolsets and would impact the size of the traces and the time to perform a simulation. Similarly, the environment does not model memory-mapped files. It is an open research question how to build a system to do prefetching and cache management for VM and mapped files. Nonetheless, the goal of this study is to understand informed prefetching and cache management of the disk cache, orthogonal to issues of virtual memory, so I assume that sufficient memory exists to run our experiments without excessive paging. In future work, when the algorithms are extended to include virtual memory, the simulation environment will have to be extended similarly.

Next, the traces do not capture process compute time spent initiating I/Os. The simulator models this time, simulating the necessary computation whenever an I/O is submitted, but any CPU time that the traced processes spent initiating accesses is not subtracted from the script records. This has the effect of slightly dilating the process CPU time for accesses that caused disk I/O on the original system.

The current system does not capture meta-data reads. Although there are a number of these reads, only a small fraction do not hit in the cache (*i.e.*, one indirect block for each 256 data blocks for sequential reads, or one quarter of a percent overhead).

The simulator does not model system call overhead. Thus, a program that reads two extents from a single file system block using two system calls is treated as equivalent to a program that makes a single system call.

I also do not model the standard background processes, interrupts, and other activities that exist on a Unix platform and occasionally require the CPU or the disk.

## 3.2 Applications

This section describes the benchmark suite of six I/O-intensive applications, as modified to disclose their accesses. The particular applications were chosen to represent a broad range of problem domains. The changes made to these applications to allow them to generate hints (*i.e.*, all the work described in Section 3.2) were done by Hugo Patterson and the other authors of [PGG+95]. The text of this section is largely drawn from that paper.

### 3.2.1 Davidson

The Multi-Configuration Hartree-Fock, MCHF, is a suite of computational-physics programs which we obtained from Vanderbilt University, where they are used for atomic-physics calculations. Davidson [SF94] is an element of the suite that computes, by successive refinement, the extreme eigenvalue-eigenvector pairs of a large, sparse, real, symmetric matrix stored on disk. In our test, the size of this matrix is 16.3 MByte.

Davidson iteratively improves its estimate of the extreme eigenpairs by computing the extreme eigenpairs of a much smaller, derived matrix. Each iteration computes a new derived matrix by a matrix-vector multiplication involving the large, on-disk matrix. Thus, the algorithm repeatedly accesses the same large file sequentially. Annotating this code to give hints was straightforward: at the start of each iteration, Davidson discloses the whole-file sequential read anticipated in the next iteration.

### 3.2.2 Xds

XDataSlice (Xds) is an interactive scientific visualization tool developed at the National Center for Supercomputer Applications at the University of Illinois [Nat89]. Among other features, Xds lets scientists view arbitrary planar slices through their 3-dimensional data with a false color mapping. The datasets may originate from a broad range of applications such as airflow simulations, pollution modeling, or magnetic resonance imaging, and tend to be very large.

It is often assumed that because disks are so slow, good performance is only possible when data is in main memory. Thus, many applications, including Xds, require that the entire dataset reside in memory. Because memory is still expensive, the amount available often constrains scientists who would like to work with higher resolution images and therefore larger datasets. Informed prefetching invalidates the slow-disk assumption and makes out-of-core computing practical, even for interactive applications. To demonstrate this, we added an out-of-core capability to Xds.

To render a slice through an in-core dataset, Xds iteratively determines which data point maps to the next pixel, reads the datum from memory, applies false coloring, and writes the pixel in the output pixel array. To render a slice from an out-of-core dataset, Xds splits this loop in two. Both to manage its internal cache, and to generate hints, Xds first maps all of the pixels to data-point coordinates and stores the mapping in an array. Having determined which data blocks will be needed to render the current slice, Xds ejects unneeded blocks from its cache, gives hints to TIP, and reads the needed blocks from disk. In the second half of the split loop, XDS reads the cached pixel mappings, reads the corresponding data from the cached blocks, and applies the false coloring [PG94].

Our test dataset consists of $512^3$ 32-bit floating point values requiring 512 MByte of disk storage. The dataset is organized into 8 KByte blocks of 16x16x8 data points, and is stored on the disk in Z-major order. Our test renders 25 random slices through the dataset.

### 3.2.3 GNULD

GNULD version 2.5.2 is the Free Software Foundation's object code linker which supports ECOFF, the default object file format under OSF/1. GNULD performs many passes over input object files to produce the output linked executable. In the first pass, GNULD reads each file's primary header, a secondary header, and its symbol and string tables. Hints for the primary header reads are easily given by replicating the loop that opens input files. The read of the secondary header, whose location is data dependent, is not hinted. Its contents provide the location and size of the symbol and string tables for that file. A loop splitting technique similar to the technique described above for the XDS application is used to hint the symbol and string table reads.

After verifying that it has all the data needed to produce a fully linked executable, GNULD makes a pass over the object files to read and process debugging symbol information. This involves up to nine small, non-sequential reads from each file. Fortunately, the previously read symbol tables determine the addresses of these accesses, so GNULD loops through these tables to generate hints for its second pass.

During its second pass, GNULD constructs up to five shuffle lists which specify where in the executable file object-file debugging information should be copied. When the second pass completes, GNULD finalizes the link order of the input files, and thus the organization of non-debugging ECOFF segments in the executable file. GNULD uses this order information and the shuffle lists to give hints for the final passes.

Our test links the 562 object files of the TIP-1 kernel, an earlier version of the TIP2 kernel described here. These object files comprise approximately 64 MByte, and produce an 8.8 MByte kernel.

### 3.2.4 POSTGRES1 and POSTGRES2

Postgres version 4.2 [SR86, SRH90] is an extensible object-oriented relational database system from the University of California at Berkeley. In our test, Postgres executes a join of two relations. The outer relation contains 20,000 unindexed tuples (3.2 MByte) while the inner relation has 200,000 tuples (32 MByte) and is indexed (5 MByte). We run two cases. In the first (POSTGRES1), 20% of the outer relation finds a match in the inner relation. In the second (POSTGRES2), 80% find a match. One output tuple is written sequentially for every tuple match.

To perform the join, Postgres reads the outer relation sequentially. For each outer tuple, Postgres checks the inner relation's index for a matching inner tuple and, if there is one, reads that tuple from the inner relation. From the perspective of storage, accesses to the inner relation and its index are random, defeating sequential lookahead, and have poor locality, defeating caching. Thus, most of these inner-relation accesses incur the full latency of a disk read.

To disclose these inner-relation accesses, we employ a loop-splitting technique similar to that used in XDS and GNULD. In the pre-computation phase, Postgres reads the outer relation (disclosing its sequential access), looks up each outer-relation tuple address in the index (unhinted), and stores the addresses in an array. Postgres then discloses these pre-computed block addresses to TIP. In the second pass, Postgres rereads the outer relation without hints but skips the index lookup and instead directly reads the hinted inner-relation tuples whose addresses are stored in the array.

### 3.2.5 SPHINX

SPHINX [LHR90] is a high-quality, speaker-independent, continuous-voice, speech-recognition system. In our experiments, SPHINX recognizes an 18-second recording commonly used in SPHINX regression testing.

SPHINX represents acoustics with Hidden Markov Models and uses a Viterbi beam search to prune unpromising word combinations from its current Viterbi search graph. To achieve higher accuracy, SPHINX uses a language model to effect a second level of pruning. The language model is a table of the conditional probability of word-pairs and word-triples. At the end of each 10 ms acoustical frame, the second-level pruner is presented with the words likely to have ended in that frame. For each of these potential words, the probability of it being recognized is conditioned by the probability of it occurring in a triple with the two most recently recognized words, or occurring in a pair with the most recently recognized where there is no entry in the language model for the current triple. To further improve accuracy, SPHINX makes three similar passes through the search data structure, each time restricting the language model based on the results of the previous pass.

SPHINX (like XDS) came to us as an in-core only system. Since it was commonly used with a dictionary containing 60,000 words, the language model was several hundred megabytes in size. With the addition of its internal caches and search data structures, virtual memory paging occurs even on a machine with 512 MByte of memory. We modified SPHINX to fetch from disk the language model's word-pairs and word-triples as needed. This enables SPHINX to run on our 128 MByte test machine 90% as fast as on a 512 MByte machine.

We also modified SPHINX to disclose the word-pairs and word-triples needed to evaluate each of the potential words offered at the end of each frame. Because the language model is sparsely populated, at the end of each frame there are about 100 byte ranges that must be consulted, of which all but a few are in SPHINX's internal cache. However, there is a high variance on the number of pairs and triples consulted and fetched, so storage parallelism is appropriate.

### 3.2.6 AGREP

AGREP, a variant of grep, was written by Wu and Manber at the University of Arizona [WM92]. It is a full-text pattern matching program that performs approximate matches. Invoked in its simplest form, it opens the files specified on its command line one at a time, in argument order, and reads each sequentially.

Since the arguments completely determine the files that will be accessed, and the access order, AGREP can issue hints for all accesses upon invocation.[1] AGREP simply loops through the argument list and informs the file system of the files it will read. When searching data collections such as software header files or mail messages, hints from AGREP frequently specify hundreds of files too small to benefit from history-based readahead, such as OBL. In such cases, informed prefetching has the advantage of being able to prefetch across files and not just within a single file.

In our benchmark, AGREP searches 1349 kernel source files occupying 1922 disk blocks for a simple string that does not occur in any of the files.

## 3.3 Generating the Traces

The suite of applications described above was run on a Digital 3000/600 workstation containing a 175 MHz Alpha (21064) processor, 128 MByte of memory, and a single HP C2247 1 GByte disk attached via a fast SCSI-2 adapter.

[1]This is not true for some of Agrep's options; our test uses the default exact matching.

### 3.3.1 Tracing Tool

We modified the Digital Unix 3.2g-3 kernel[2] to trace read, write, open and close system calls, as well as hint delivery and context switches. Each tracing call logged relevant information for the particular system call (*e.g.*, byte range read and vnode number for files) as well as a standard set of boilerplate information (task ID and time in cycles) into a statically allocated buffer set to 20 MByte for these runs. The kernel buffer was then mapped to user space and printed in a human-readable form.

We post-processed the traces into a script file capable of driving the simulator, replacing vnode numbers and byte ranges with logical disk blocks, and real time values with process times. This code represented about 300 lines of Perl, and used a utility written by Hugo Patterson to generate the "map" of a file: the location of each file block in the file's disk partition (or in other words, the mapping from file block to disk block). The file maps were generated statically, rather than on-the-fly; thus, we did not capture maps for a small number of temporary files written and destroyed by the applications. These files were assumed to lie at a random location on the disk, but did not account for a significant fraction of the operations.

## 3.4 Discussion of Traces

This section describes the traces themselves. Table 3.1 shows the balance of reads, writes and hints for each trace. All the applications are read-dominated, though there are some writes occurring throughout the traces. The trace-collection environment captures system calls, so the numbers in Table 3.1 do not always capture the number of blocks read.[3] Table 3.2 corrects for this fact, and shows how much actual data is read by each trace. Table 3.3 gives more information about the hints themselves.

Section 4.8.1 describes the hint-tracking mechanism used by all the algorithms. In the upcoming sections I describe how hints and their corresponding reads are interspersed through each trace, giving more individual details and presenting "profiles" of the access patterns. In these profiles, hinting regions are labeled "hint" or "H" as space permits, and regions of consumption (read or write) are labeled "consume", "con" or "C". The profiles are indexed by record number in the script file.

[2]The modifications to the kernel necessary to collect and post-process these traces (everything in Section 3.3.1, unless specifically noted otherwise), was done by David Rochberg.

[3]For instance, several systems calls in a row may access successive byte ranges in the same disk block. Also, most applications determine end-of-file by noting that a read returns 0 bytes; the traces do not capture return values and therefore include an additional read call per file.

| Trace | Reads | Writes | Hints | Total |
|---|---|---|---|---|
| DAVIDSON | 133656 | 1403 | 127429 | 262488 |
| XDS | 46348 | 0 | 45241 | 91589 |
| GNULD | 18348 | 2621 | 14106 | 35075 |
| POSTGRES1 | 8695 | 141 | 4455 | 13291 |
| POSTGRES2 | 31264 | 522 | 16325 | 48111 |
| SPHINX | 77428 | 24 | 74700 | 152152 |
| AGREP | 4270 | 0 | 2921 | 7191 |

Table 3.1: Breakdown of Traces by Operation.

| Trace | Trace "read" records | Actual blocks read | Distinct Blocks Read |
|---|---|---|---|
| DAVIDSON | 133656 | 125491 | 2170 |
| XDS | 46348 | 45421 | 32622 |
| GNULD | 18348 | 12482 | 7202 |
| POSTGRES1 | 8695 | 8658 | 3737 |
| POSTGRES2 | 31264 | 31227 | 5207 |
| SPHINX | 77428 | 30856 | 20716 |
| AGREP | 4270 | 2922 | 2922 |

Table 3.2: Actual Read Statistics: The tracing environment captures separate reads to multiple byte ranges within the same block as distinct operations. The first column above shows the number of such operations. The second column shows the number of reads when back-to-back accesses to the same block are collapsed. The third column shows the number of reads when all accesses to the same block are collapsed.

### 3.4.1 DAVIDSON

The DAVIDSON trace repeatedly reads a large file (2089 8 KByte blocks) sequentially. Hints are given before each batch of reads. This behavior is shown in Figure 3.1, which shows the profile of a prefix of the entire trace. The initial segments are longer — the implementation hints a batch, then hints two more batches simultaneously,[4] consumes two batches, and then settles into a pattern of hinting and consuming a batch at a time. As Table 3.1 shows, there is a full batch of hints remaining at the end of the run.[5] The

[4]Hinting an extra batch at the beginning turned out to be the easiest way to avoid running out of hints at the end of each batch.

[5]During simulation, different algorithms reacted differently to the final batch of unused hints. In order to eliminate this extraneous effect, the actual trace used in the simulations does not present the

| Trace | Missing Hints | Bad Hints | Unused Hints | Consumed Hints | Total Hints |
|---|---|---|---|---|---|
| Davidson | 8316 | 0 | 2089 | 125340 | 127429 |
| Xds | 1057 | 0 | 0 | 45241 | 45241 |
| Gnuld | 4357 | 116 | 0 | 13990 | 14106 |
| Postgres1 | 4205 | 71 | 0 | 4384 | 4455 |
| Postgres2 | 15072 | 242 | 0 | 16083 | 16325 |
| Sphinx | 1895 | 0 | 0 | 74700 | 74700 |
| Agrep | 1348 | 0 | 0 | 2921 | 2921 |

Table 3.3: Hint Accuracy: The first numerical column shows missing hints. The next three columns are a breakdown of all hints into "Bad" (no read was found corresponding to the hint, and the hint was dropped to keep the hint sequence on track with the access sequence), "Unused," (the hint was still waiting for a corresponding read when the program ended), and "Consumed" (the hint matched a read). The last column gives the total number of hints delivered.

hint/consume of a single batch at a time continues without exception through the entire trace until the very end, where the approximately 1400 writes occur. The computation requires 60 iterations to converge, so each piece of data is read exactly 60 times. However, occasionally multiple pieces of data in the same block are read in different system calls, so the total number of reads in the script is slightly larger than expected. Finally, there are 139 additional reads before the actual computation begins.

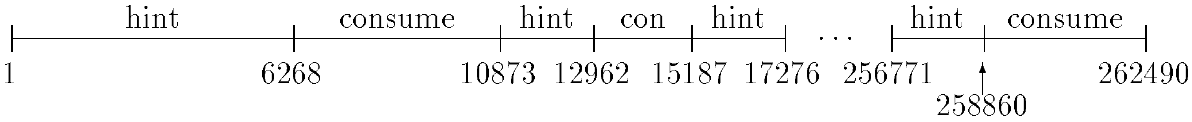



Figure 3.1: Davidson Profile

### 3.4.2 Xds

The profile for a subsection of the Xds trace is shown in Figure 3.2. The actual profile contains 51 segments: 25 hints and consumptions for the 25 random slices rendered in the workload, plus a short initial period of unhinted consumption. Xds touches 45,421 blocks in the course of execution, an average of 1817 per slice; of these, 32,622 are distinct. The

very last batch of hints.

| Number of independent accesses | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| Number of blocks | 22755 | 7575 | 1885 | 346 | 52 | 7 |

Table 3.4: Re-use Characteristics of XDS: There were 22755 blocks that were used only once (no re-use), 7575 blocks that were used twice, and so on.

re-use characteristics are given in Table 3.4. As the table shows, the number of blocks experiencing a particular level of re-use drops off by a factor of at least three from blocks used $n$ times to blocks used $n+1$ times. In general, the trace exhibits very little re-use. The entire file is 512 MBytes, and the 25 slices in the trace touch 49.7% of the file.

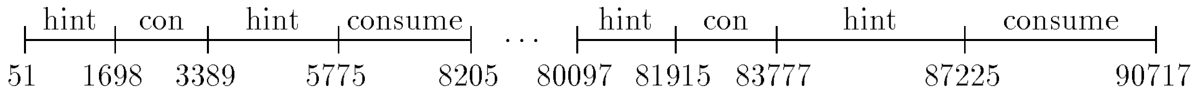



Figure 3.2: XDS Profile

### 3.4.3 GNULD

Figure 3.3 gives the profile for GNULD. The early mixed section contains the small batches of hints and reads that are given in the first stages of the process. Hints are then generated for the second pass through the object files, and then when the link order is fixed, the final batch of hints is given for the final stages of the process.

Re-use occurs roughly as follows. There is essentially no re-use during the first 6177 lines, through the end of the first consumption. During the second, larger consumption segment, about 5,000 new blocks are touched, along with 1,200 of the 1,700 blocks already touched. Finally, during the third consumption segment, over about 4,300 reads, only about 340 new blocks are touched and the rest of the reads are to already-seen blocks. Overall, there are 12482 reads to 7202 distinct blocks.

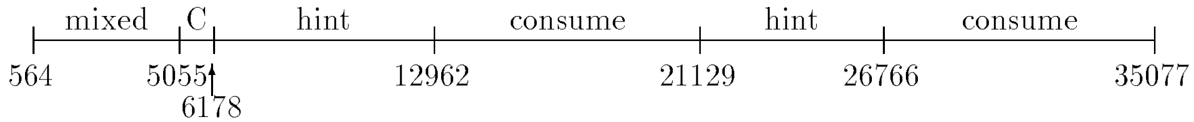



Figure 3.3: GNULD Profile

### 3.4.4 POSTGRES1

Figure 3.4 shows the profile for the POSTGRES1 trace, in which 20% of the outer relation finds a match in the inner relation. The initial hints are for the outer relation, which is read sequentially. The first consumption segment is for the pre-processing phase in which the outer relation tuples cause accesses to the inner relation's index. The results of this phase are stored in an array then provided as hints during the hinting phase. The final consumption phase re-reads the outer relation and uses the hints to access the data of inner relation itself. Although only 20% of the outer relation finds a match in the inner relation, the resulting 4,000 matches touch 2688 of 4096 blocks (32 MByte) in the inner relation as the distribution of these 4,000 matches is essentially random across the blocks of the inner relation. The outer relation is 409 blocks, and the index for the inner relation is 640 blocks. The outer relation blocks are read once from hints during the first pass, and then again without hints during the second pass. The inner relation index blocks are not hinted but experience substantial re-use. The inner relation data blocks are hinted but do not experience much re-use, especially compared to the POSTGRES2 trace. Essentially, the outer relation blocks are used twice, the index blocks are used more than twice (though re-use drops off quickly — no block is used more than 14 times), and the data blocks are used once.

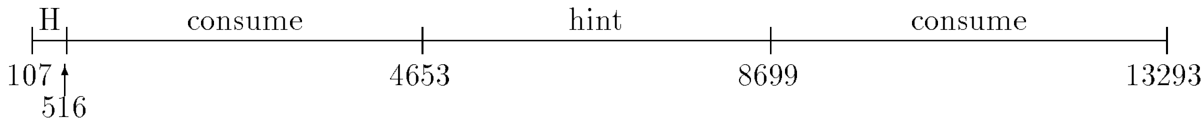



Figure 3.4: POSTGRES1 Profile

### 3.4.5 POSTGRES2

Figure 3.5 shows the profile for the POSTGRES2 trace, in which 80% of the outer relation matches in the inner relation. The initial hinting segment has the same length in this trace as in the POSTGRES1 trace, but the remainder of the segments are relatively much larger in POSTGRES2 since the search accesses many more of the 200,000 elements of the inner relation. 16,000 of the 20,000 tuples in the outer relation find a match in the inner relation; this results in reads to essentially all of the 32 MByte of inner relation blocks. The situation has changed slightly from the POSTGRES1 case because there is now more re-use of the inner relation data blocks. In fact, the average block is used 6 times in the course of the trace, and some data blocks are touched as many as 45 times.

| Hints in Batch | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | > 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Num Batches | 189 | 103 | 73 | 57 | 36 | 26 | 14 | 12 | 12 | 11 | 211 |

Table 3.5: Histogram showing number of batches of various sizes in SPHINX trace.

Thus, the hints to the inner relation data blocks are useful for both prefetching and cache management.


Figure 3.5: POSTGRES2 Profile

### 3.4.6 SPHINX

The profile for a subsection of the SPHINX trace is show in Figure 3.6. The trace takes about 154 seconds. During the first 52 seconds, SPHINX reads a large dictionary file sequentially, touching 19805 blocks exactly once per block. During the remaining 102 seconds, SPHINX performs recognition using a Viterbi beam search, reading 11051 blocks, 3308 of which are distinct. Thus, re-use is substantial during the search phase, but not during the initial large read. Table 3.6 shows a histogram of re-use during the search phase. The application is unique in our suite in that it sometimes provides extremely shallow hint queues, because its accesses are highly dynamic and depend on decisions made in the recent past. During the recognition phase the application passes hints to the system every 10 ms disclosing the elements of the language model that will be required during the next round. The number of accesses made during these alternating rounds of hints and consumption has high variance, and is often quite small. Table 3.5 gives a histogram showing how many batches of $n$ hints SPHINX discloses to the system, for various values of $n$.

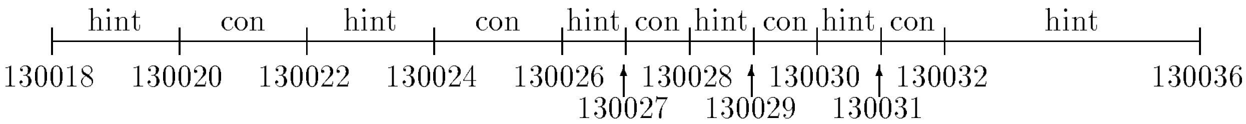



Figure 3.6: SPHINX Profile (subsection)

| Number of Accesses | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | > 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Number of Blocks | 942 | 563 | 503 | 459 | 193 | 262 | 212 | 134 | 14 | 8 | 18 |

Table 3.6: Re-use characteristics of SPHINX: The histogram shows number of blocks that are accessed $n$ times, for various values of $n$, in the search phase of the SPHINX trace. The first column says that 942 blocks were accessed exactly once, and so on.

### 3.4.7 AGREP

The profile for the AGREP trace is given in Figure 3.7. In the initial hinting segment complete hints are given for all files on the command line; AGREP then searches sequentially through these files during the consume segment. All read system calls are for an entire block; 2922 blocks are read, and no block is read more than once.

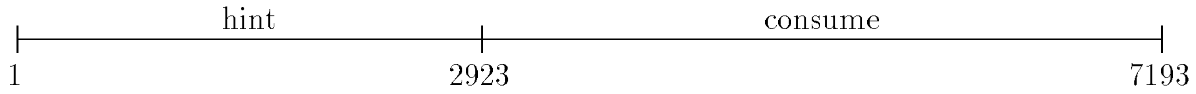



Figure 3.7: AGREP Profile

## 3.5 The Simulator

The simulator is built on top of the Berkeley RaidSim simulator [Lee89, CP90, LK91], as modified by Mark Holland [Hol94] at CMU. RaidSim can simulate various flavors of RAID disk arrays using an internal geometry-aware disk simulator to determine disk access times. In the experiments described below, data is striped over an array of disks (from 1–10 disks), with no parity and a stripe unit of eight 8 KByte blocks (64 KByte). The disk simulator, described below in Section 3.6, models the HP97560 disk drive[RW94]. The simulator also supports various forms of disk queueing; in our experiments all issued accesses are CSCAN sorted in the disk queues. RaidSim was modified to include a buffer cache module layered on top of the disk array, and modules for all the algorithms. The simulator represents about 40000 lines of C code, about half of which represents my additions.

Earlier work in the Parallel Data Lab had added a module to drive RaidSim from traces instead of relying on randomly generated workloads; I drive the simulator with the traces described above. As mentioned earlier the tracing environment includes all application accesses, and captures process computation time before each access. RaidSim's support for multiple threads, and integrated CPU scheduler, allows the concurrent simulation of multiple separately-scripted processes. Thus, arbitrary collections of individual processes can run side-by-side in the simulator.

### 3.5.1 Tracing an I/O Through the Simulator

To give an intuition for the processing of a typical read request, here is a high-level description of the path a demand miss takes through the system. As the simulator is designed to Manage Application Disclosures for Caching And Prefetching, I will refer to it as MADCAP.

1. MADCAP reads the next record from a script file, parses it, and extracts the device, block number, size, pid, and inter-access compute time.
2. A simple hint-tracking algorithm finds the hint corresponding to this read, if any, and updates the application's current position in the hint sequence.
3. MADCAP looks for the block in the cache. If a block is found and the I/O has completed, the block is returned. If a block is found but its I/O is pending, a wait occurs. Assume instead that the requested block is not found in the cache.
4. An algorithm-specific getblk routine is called to allocate a buffer for the fetch. This may involve waiting if no buffers are available. The resulting block is removed from the cache hash table and its busy flag is set. Any hints for the block about to be evicted are marked, and any other data structures that must be updated are notified.
5. A new thread is spawned to perform the I/O. Since the demand read is synchronous, the calling thread blocks waiting for completion. The child thread sends the request to the disk driver, which inserts it into a queue sorted by the active disk scheduling discipline. The child then waits for the I/O to complete.
6. The disk subsystem processes the I/O in turn, logs appropriate status, and sets the appropriate flags in the block header.
7. When the child completes, the parent awakes, turns off the buffer's busy flag, and returns the block to the hash table. Any reordering of the LRU queue, or other tracking that the prefetching algorithm must perform, is done here. Also, future hints for this block for marked as being in core.
8. Finally, the prefetcher is given the opportunity to issue new prefetches.

| | |
|---|---|
| Capacity | 1.3 GByte |
| Cylinders | 1935 |
| Size | 5.25in |
| Rotational Speed | 4002 RPM |
| Average 8 KByte Access | 22.8ms |
| Host Interconnect | SCSI-2 |
| Interconnect Speed | 10 MByte/S |

Table 3.7: The HP 97560 Disk Drive

## 3.6 The Disk Simulator

This section describes the disk drive simulator itself. There were several options for generating an accurate disk model. The first was to use the existing model from RaidSim [Lee89], which had been modified substantially by Mark Holland [Hol94]. Using this model was unattractive because the existing code did not include an on-disk readahead cache, and work of Ruemmler and Wilkes [RW94] has shown that disk caching can have a significant effect on model accuracy. In fact, their model is accurate to within a demerit of 5.7% (see below for the definition of this measure), but without caching the demerit grows to 112%. Also, the existing model did not have CSCAN queueing, which is commonly used and has been shown effective on modern drives [SCO90].

The next option was to incorporate an off-the-shelf public-domain disk model. In particular the model of Kotz *et al.* [KTR94] is widely used. I decided not to take this approach because incorporating a new disk model into RaidSim, and matching the thread interfaces, seemed like a significant project.

Instead, I implemented readahead and CSCAN queueing, and then validated the resulting system again the Kotz model. Following Kotz, I simulate the HP 97560 disk drive. The characteristics of this drive are shown in Table 3.7. The validation was done as follows. I ran the simulator on the suite of traces described above without CPU activity, and logged requests as they arrived at the disk under whatever scheduling discipline was active (typically CSCAN). The logs of disk activity included logical disk block numbers and request lengths. I then post-processed the log files to produce scripts that could drive Kotz's simulator, and compared the output of their system to mine.

Ruemmler and Wilkes suggest *demerit* as an appropriate figure of merit for this type of comparison. Given two sequences of I/O operations with durations, the sequences are sorted by request duration, and the squares of the differences between corresponding durations are computed. The demerit is the square root of the average of these values, expressed as a percentage of the mean I/O time. Graphically, this corresponds to the fol-

| Application | Average I/O Time | Demerit |
|---|---|---|
| Xds | 6.5ms | 2.85% |
| Gnuld | 10ms | 3.45% |
| Agrep | 12.7ms | 3.65% |
| Postgres2 | 17ms | 2.97% |
| Davidson | 4.7ms | 3.23% |
| Postgres1 | 16.6ms | 2.78% |
| Sphinx | 5.7ms | 4.5% |
| Average | 10.5ms | 3.3% |

Table 3.8: Average I/O Times for our simulations and demerit figures for our simulations versus Kotz's diskmodel.

lowing intuition. Picture a graph of I/O time versus fraction of requests completed within that time, so a profile for a particular run will begin at (0,0) and move to ($max\_io\_time$,1). The demerit is the rms of the *horizontal* distance between two such curves, expressed as a fraction of the average.

Table 3.8 gives the demerit figure for my simulator versus Kotz's on each trace.

### 3.6.1 Features Missing from the Disk Model

The Kotz model contains several features that I do not model, including detailed SCSI bus simulation (I model the bus as a constant-latency overhead; Kotz *et al.* include a model of bus contention), read and write fences, and status messages. As none of the applications consume data at the bus bandwidth of 10 MByte/sec, these effects are minimal. However, my data layout model does not include sparing so while the relative locations of neighboring blocks will be accurate, the actual mapping of physical to virtual sectors will not be exact. I expect this to introduce a small amount of noise into the results, and in fact Table 3.8 shows that the models are not identical, but are extremely close. The average demerit percentage of 3.3% means that my model is closer to Kotz *et al.*'s model than their model is to the actual drive. In conclusion, inaccuracies in the disk model may result in small inaccuracies in the reported overall execution times, but will not alter the conclusions.

# Chapter 4

# Algorithms

*It was a hack like any other, only a trifle dirtier.*

*— Henry James, "The American," 1875*

This chapter describes four informed prefetching and caching algorithms, presented in a combination of functional and temporal ordering. Before this work two systems had been implemented: the TIP2 system of Patterson, Gibson *et al.* [PGG+95] and the LRU-SP/AGGRESSIVE system of Cao *et al.* [CFKL95b] (this algorithm was embedded in a filesystem called ACFS, application-controlled file system, described in Pei Cao's thesis [Cao96]). Recall that the SPACE problem (Standalone Prefetching And Cache Eviction) involves managing prefetching and cache eviction decisions for a single process with full knowledge of future accesses and a fixed set of resources. The allocation problem, the other piece of the puzzle, involves partitioning resources among multiple competing processes that may be disclosing arbitrary fractions of their resources. TIP2 and LRU-SP/AGGRESSIVE take different approaches to both problems. I begin by describing these two systems. Next, I focus on the SPACE problem, comparing the approaches taken by these two systems, and describing collaborative work with Kimbrel, Karlin, Patterson, Gibson, Cao, Bershad, Felten and Li [KTP+96]. TIP2 is limited in how deeply it will prefetch ahead in to the request stream, while LRU-SP/AGGRESSIVE prefetches as deeply as resources allow. Our collaboration focused on determining how deeply to prefetch in any situation — the resulting algorithm, FORESTALL, makes decisions based on a dynamic estimate of disk load. Next I consider improving TIP2 and LRU-SP/AGGRESSIVE by incorporating dynamic prefetching according to FORESTALL's load estimate. Extending TIP2 requires substantial groundwork, but the resulting algorithm, TIPTOE (TIP with Temporal Overload Estimators), performs better on average than the other approaches across a wide range of experiments. Extending LRU-SP/AGGRESSIVE, yielding LRU-SP/FORESTALL, is more straightforward. I present these two new algorithms and then discuss implementation details.

## 4.1 TIP2

The TIP2 system of Patterson, Gibson *et al.* [PGG+95] presents an integrated approach to the SPACE problem and the allocation problem. I begin with a two-paragraph summary of the approach, and then give details.

In the single-process model, TIP2 prefetches conservatively due to its system model, which assumes large disk arrays and no disk queueing. In such a model, fetches go directly to disk and return within some bounded time. In addition, there is a lower limit on the rate of application consumption since, even with no inter-access computation and no cache misses, there is non-negligible overhead in processing a cache hit (servicing the system call, copying the data to user space, and so on). Therefore there is no advantage to prefetching further ahead than the fixed number of accesses necessary to guarantee that the prefetch will complete in time. TIP2's SPACE algorithm has been referred to elsewhere as FIXED-HORIZON, since prefetches are submitted only for missing blocks within a fixed "prefetch horizon."

TIP2's approach to the allocation problem, *cost-benefit analysis*, is a powerful and general tool that has been successful in other domains as well — Ng, Faloutsos and Sellis [NFS91] apply the technique to database cache management. To serve multiple processes, each disclosing an arbitrary fraction of its accesses, the buffer cache manager must cache two distinct types of data. First, it must maintain a traditional LRU cache, caching blocks that have been read recently in case unhinted accesses arrive for the same blocks. Second, it must maintain a hinted cache, holding buffers for which future hints exist. When a buffer is needed, for a prefetch or a demand read, it must be taken from one of these sources. Cost-benefit analysis maintains *estimators* to compute the cost of evicting each block, and the benefit of submitting a prefetch or demand read. Whenever the greatest benefit is larger than the lowest cost, an I/O is initiated and a reallocation occurs.

### 4.1.1 System Model

TIP2's system model assumes that all application I/O accesses request a single file block that can be read in a single disk access; that system parameters such as disk access latency, $T_{\text{disk}}$, are constants; and that there is enough disk parallelism for there never to be any congestion (that is, there is no disk queueing) so every fetch returns in $T_{\text{disk}}$ time. Let $T_{\text{cpu}}$ be the inter-access application compute time; $T_{\text{hit}}$ be the time to read a block from the cache; and $T_{\text{driver}}$ be the computational overhead of allocating a buffer, queueing the request at the drive, and servicing the interrupt when the disk operation completes. $T_{\text{miss}}$, the time to service a demand miss, is then $T_{\text{miss}} = T_{\text{hit}} + T_{\text{driver}} + T_{\text{disk}}$.

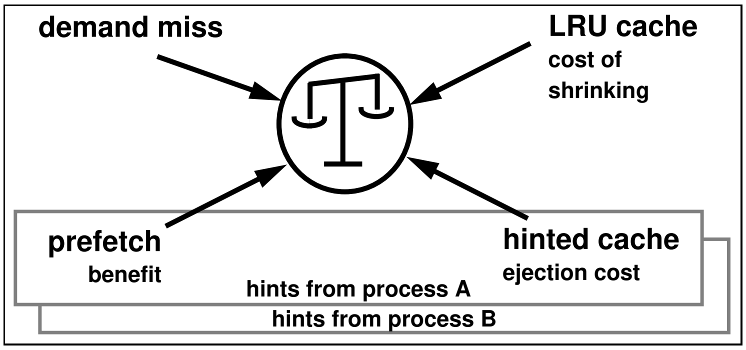



Figure 4.1: TIP2's informed cache manager schematic. Independent estimators express different strategies for reducing I/O service time. Demand misses need a buffer immediately to minimize the stall that has already started. Informed prefetching would like a buffer to initiate a read and avoid disk latency. To respond to these buffer requests, the buffer allocator compares their estimated benefit to the cost of taking a buffer from a buffer supplier. The LRU queue caches blocks for unhinted accesses. Informed caching holds on to the blocks that will be re-accessed soonest. The buffer allocator takes the least-valuable buffer held by any supplier to fulfill a buffer demand when the estimated benefit exceeds the estimated cost.

### 4.1.2 Cost-Benefit Analysis

Figure 4.1 gives an overview of TIP2's cost-benefit cache manager. The cache manager's goal is to allocate resources to minimize *I/O service time*, the time it takes a read or write system call to complete (note that this includes computational overhead as well as I/O stall time). The managed resources are disks and file cache buffers. The consumers of these resources are demand accesses that miss in the cache and prefetches of hinted blocks. Buffers are supplied by the LRU cache and the hinted cache.

The resource manager must decide whether reallocating a buffer from a supplier to a consumer to initiate an I/O will reduce I/O service time. The system estimates the benefit of using a buffer to initiate a disk access and the cost of taking a buffer from a buffer supplier. The buffer allocator continually compares these estimates and reallocates buffers when doing so would reduce I/O service time.

For different estimates to be comparable, they must be expressed in the same terms. Patterson, Gibson *et al.* therefore define a *common currency* for the expression of cost and benefit estimates that relates I/O service time to the employment of cache resources over time. The unit of buffer usage is the occupation of one buffer for one inter-access period, called one *buffer-access*, or one unit of *bufferage*. The common currency is the *magnitude of the change in I/O service time per buffer-access.*

### 4.1.3 TIP2 Estimators

When an application suffers a demand miss, consumption halts until the data arrives; therefore the benefit for using a buffer to service a demand miss, in terms of change in I/O service time per buffer-access, is infinite:

$$\text{Benefit}(\text{demand\_miss}) = \infty.$$

The benefit for initiating a prefetch is much less clear. TIPTOE, described below, devotes great effort to improving this estimator, so I will provide only a brief description of TIP2's approach. Earlier I gave an intuition that, under TIP2's system model, it is never necessary to prefetch more than a fixed number of accesses ahead into the hint stream; I now make this notion more concrete. Even if all the blocks in a hinted sequence are cached, consuming each block will require some application computation, $T_{\text{cpu}}$, plus the time to read each block from the cache, $T_{\text{hit}}$. In the most conservative estimate, $T_{\text{cpu}}$ may be 0 but $T_{\text{hit}}$ time is still required to consume each block. Under the assumption of no disk queueing, a prefetch will always complete $T_{\text{disk}}$ time after it is submitted. If, for instance, $T_{\text{hit}} = 1$ and $T_{\text{disk}} = 50$, a prefetch submitted 50 accesses before it is needed will always complete in time. More generally, if $\hat{P} = T_{\text{disk}}/T_{\text{hit}}$, an application need never submit a prefetch more than $\hat{P}$ accesses in advance. $\hat{P}$ is called the *prefetch horizon.*

The benefit of prefetching a missing block $x$ accesses in the future is therefore 0 whenever $x \geq \hat{P}$. When $x < \hat{P}$, any particular I/O may stall, but that stall may be amortized across other requests. Patterson and Gibson present a pipeline model of parallel I/O's, and derive the steady-state decrease in stall that results if prefetches are submitted $x$ accesses in advance rather than $x - 1$ accesses in advance. I omit the derivation, and give the final equation:

$$\text{Benefit}(\text{prefetch}, x) = \begin{cases} T_{\text{disk}} & x = 0 \\ \frac{T_{\text{disk}}}{x(x+1)} & x < \hat{P} \\ 0 & x \geq \hat{P} \end{cases}.$$

Next, I describe the cost estimators. TIP2 profiles the effectiveness of the LRU cache, maintaining an estimate $H(n)$ of the hit rate of a cache of $n$ pages, for all values of $n$ up to the total number of buffers in the system. Since the actual size of the LRU cache is usually smaller than the total number of pages, *ghost buffers* that are not associated with a physical page are used for the remainder. Using these ghost buffers, the profiler remembers how many hits occur at each distance into the LRU list; this information is sufficient to compute $H(n)$ for all $n$.

Given this estimate, the cost of shrinking the LRU from $n$ pages to $n-1$ pages depends on the change in hit rate: $\Delta H(n) \stackrel{def}{=} H(n) - H(n-1)$. If removing a page from the LRU decreases the hit rate of unhinted accesses by 1% then the system will incur additional stall equal to the time to reload the page ($T_{\text{disk}} + T_{\text{driver}}$) on 1% of all unhinted accesses. So the change in I/O service time per buffer access of shrinking the LRU by one buffer will be $\Delta H(n)(T_{\text{disk}} + T_{\text{driver}})$.

However, this estimator does not capture necessary global structure in the LRU hit-rate curve. In a looping access pattern over $N$ elements, for instance, $\Delta H(n)$ is 0 for $n < N$, but the cost-benefit allocator should still try to grow the LRU to hold the working set. Therefore, TIP2 considers the benefit of growing the cache much larger than its current size in determining the marginal cost, using the following estimator:

$$\text{Cost}(\text{LRU\_eviction}, n) = \max_{i \geq n}\{\Delta H(i)\}(T_{\text{disk}} + T_{\text{driver}}).$$

Finally, I present the cost estimator for evicting data from the hinted cache. This estimator, like the estimator for benefit of prefetch, will be revisited in the discussion of TIPTOE (Section 4.5). Under TIP2's system model, if a cached block will not be needed for more than $\hat{P}$ accesses then it can be fetched back in without stall; the increase in I/O service time for doing so is $T_{\text{driver}}$. If the block will be needed $y$ accesses in the future then evicting it will provide a buffer for $y - \hat{P}$ extra accesses (since the prefetch will have to be submitted $\hat{P}$ accesses before the block is needed). So the magnitude of the change in I/O service time per buffer access is $T_{\text{driver}}/(y - \hat{P})$. Again, I will not give the details for evicting blocks that will be needed before the prefetch horizon; the complete estimator is the following:

$$\text{Cost}(\text{Hinted\_eviction}, y) = \begin{cases} T_{\text{driver}} + T_{\text{disk}} & y = 1 \\ T_{\text{driver}} + \frac{T_{\text{disk}}}{y-1} & 1 < y \leq \hat{P} \\ \frac{T_{\text{driver}}}{y-\hat{P}} & y > \hat{P} \end{cases} . \tag{4.1}$$

## 4.2 LRU-SP/AGGRESSIVE

The LRU-SP/AGGRESSIVE system maintains an explicit division between the SPACE algorithm for single-process prefetching (AGGRESSIVE) and the allocation algorithm (LRU-SP).

### 4.2.1 AGGRESSIVE

Cao, Felten, Karlin and Li [CFKL95b] present the AGGRESSIVE algorithm for single-process prefetching and caching. Like TIP2, AGGRESSIVE is designed with a specific system model in mind. Let $T_{\text{disk}}$ be the time for a disk access to complete, and $T_{\text{app}}$ be the inter-access time if the necessary block is in the cache (in the terminology of TIP2, $T_{\text{app}}$ equals $T_{\text{hit}}$ plus $T_{\text{cpu}}$ plus a fraction of $T_{\text{driver}}$ depending on how often an I/O must be submitted). In AGGRESSIVE's system model, $T_{\text{driver}}$ is 0. Let $K$ be the number of pages in the cache. Let best-fetch be the distance to the first missing block in the cache, measured in accesses, and best-evict be the distance to the first access of the block that will be needed *latest* (if some element of the cache will not be re-used, this distance will be infinite). AGGRESSIVE will evict the block best-evict accesses ahead and prefetch the block best-fetch accesses ahead exactly when the disk is free and best-fetch < best-evict (this inequality is referred to as the *do-no-harm* rule). Cao *et al.* show the following theorem about this algorithm:

**Theorem 1** *[Cao,Felten,Karlin,Li] For any request sequence, the total execution time of* AGGRESSIVE *within the system model defined above is never more than $1 + T_{\text{disk}}/(KT_{\text{app}})$ times the optimal.*

The algorithm can be modified to operate in the multi-disk domain as follows. As above, let best-evict be the distance to the best candidate for eviction. When a particular disk becomes free let best-fetch be the distance to the first missing block on that disk. Prefetch that block, evicting the block best-evict away, whenever best-fetch < best-evict. However, the theorem does not hold in this new domain. Kimbrel and Karlin present REVERSE-AGGRESSIVE, a new algorithm designed to provide theoretical guarantees in the multi-disk domain; I describe it briefly, but do not simulate it.

### 4.2.2 REVERSE-AGGRESSIVE

Our joint study found that REVERSE-AGGRESSIVE's performance was typically as good as the best of FIXED-HORIZON and AGGRESSIVE, but that it was difficult to compute the sequence of fetches and evictions, especially if hints arrive over time. FORESTALL, described below, matched REVERSE-AGGRESSIVE's performance and is much simpler to implement; therefore, I simulate FORESTALL and present it in detail. For completeness, I give a brief description of REVERSE-AGGRESSIVE— see Kimbrel and Karlin [KK96b] for details.

Briefly, REVERSE-AGGRESSIVE constructs a prefetching schedule for the *reversed* sequence that *replaces* at most one block on each disk in parallel as follows: Whenever a disk is free, determine the block $B$ not needed for the longest time on that disk. If the next request to $B$ is after the first missing block, issue a fetch for the missing block, *replacing* $B$. Transform this prefetching schedule back to a schedule for the original sequence by treating each fetch on the reverse sequence as an eviction on the forward sequence and vice versa. Kimbrel and Karlin show the following theorem:

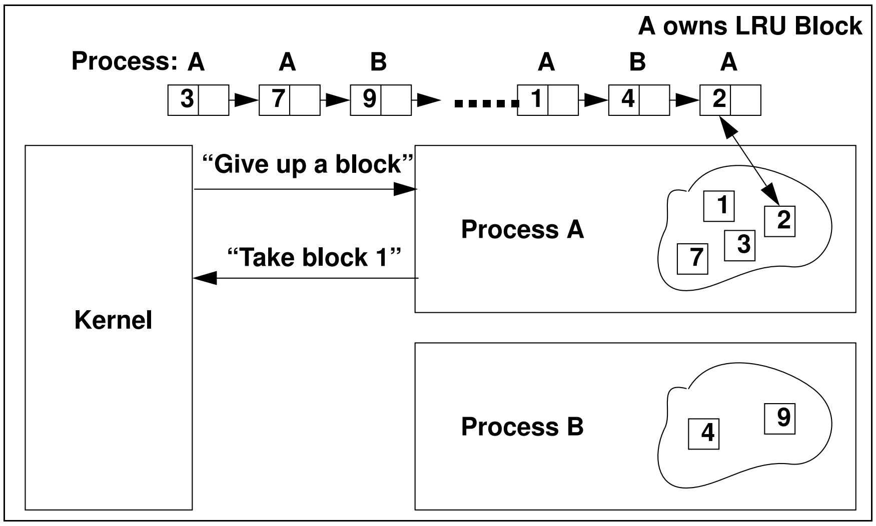



Figure 4.2: LRU-SP resource allocation algorithm. When the kernel needs a block it finds the process holding the global Least-Recently-Used block, in this case Process A. The kernel then asks Process A for a block, suggesting the LRU block. Process A may either accept the kernel's suggestion and give up block 2, or may use information not available to the kernel to choose a different block for eviction, in this case block 1. To summarize, the kernel chooses a process, then the process chooses a block.

**Theorem 2 (Kimbrel-Karlin)** *For any request sequence, and for any layout of the data on the disks, the elapsed time of* REVERSE-AGGRESSIVE *is at most* $1 + \epsilon$ *times the optimal.*[1]

### 4.2.3 LRU-SP

The LRU-SP/AGGRESSIVE system uses AGGRESSIVE to address the SPACE problem and LRU-SP to address the allocation problem. LRU-SP was developed by Cao, Felten, and Li [CFL94a, CFL94b, Cao96]; its goal is to adapt traditional LRU allocation, which has been successful due to performance and fairness qualities, to incorporate prefetching and informed cache management. LRU-SP uses a global LRU queue to partition cache buffers among the competing processes. It then applies a SPACE algorithm within each partition.

[1] $\epsilon$ here is less than $dT_{\text{disk}}/(kT_{\text{app}})$, where $d$ is the number of disks. For typical system parameters, $\epsilon$ is less than 0.1, and sometimes significantly less.

Figure 4.2 is a schematic diagram of the LRU-SP resource allocation system. When a buffer is required, either for a demand read or for a prefetch, LRU-SP finds the process owning the Least Recently Used block of the entire buffer cache[2] and asks that process if it would like to eject that block. The process may give up the LRU block itself, or may choose to evict a different block based on application-specific information. If all processes give up the block suggested by the kernel then LRU-SP becomes traditional LRU. However, if a process uses application-specific knowledge to suggest an alternate block for eviction then a difficulty arises: that process will again hold the global LRU block and would be asked to give up yet another block when the kernel requires one. To address this difficulty the kernel swaps the LRU block into the donated block's position in the LRU queue. Finally, to prevent a malicious or foolish process from mis-using this *swapping* capability to gain an unfair share of the buffer cache, a *placeholder* structure keeps track of swaps. If the donated block is re-accessed before the swapped block, the swapped block is immediately ejected. The resulting algorithm is called LRU-SP, LRU with swapping and placeholders. Note that placeholders are used only in the presence of incorrect hints — no placeholder is ever activated in our traces.

As a first approximation, LRU-SP partitions the cache among processes according to their relative rates. This is most apparent with no re-use: if process A inserts a block into the LRU list four times per second and process B one time per second then in the steady state, process A will have inserted 4/5 of the elements in the list. Section 4.7.2 examines the question for processes with re-use. In general, though, LRU-SP should perform well when faster processes derive more benefit from re-use than slower processes, but as Section 4.7.1 demonstrates, this will not always be the case.

## 4.3 A Study of Embedded SPACE Algorithms

At this point I have described TIP2 and LRU-SP/AGGRESSIVE, two full systems for multi-process prefetching and caching. Each system contains a SPACE algorithm and an allocation algorithm. In this section, I will discuss the embedded SPACE algorithms, give some intuitions about their differences, and motivate a new SPACE algorithm that combines the benefits of both earlier approaches. This work was joint between the Parallel Data Lab (Patterson, Gibson and myself), and several other researchers at Washington and elsewhere (Kimbrel, Karlin, Cao and others) [KTP+96].

Consider the TIP2 system described above running in a single-process fully-hinted domain. As there are no unhinted accesses, the value of maintaining a large LRU cache will be negligible so the entire buffer cache will be dedicated to hinted caching. The equations above for the benefit of prefetching, and the cost of evicting a block from the

[2] The owner of a block is the last process to access the block.

hinted cache, have the property that Cost(Hinted_Eviction, $x$) > Benefit(prefetch, $x$); that is, TIP2 obeys the do-no-harm rule. Since the benefit of prefetching a block more than $\hat{P}$ accesses in advance is 0, this will never happen. But for any reasonable-size buffer cache in the single-process domain, the lowest-cost block will have cost much lower than the benefit of prefetching a missing block $\hat{P} - 1$ accesses away. Thus, in this simple domain, TIP2 will submit prefetches for exactly the missing blocks within the prefetch horizon.

Our collaborative study was performed in this single-process domain; we referred to this simplification of TIP2 as FIXED-HORIZON. Similarly, in a single-process fully-hinted domain LRU-SP/AGGRESSIVE will dedicate the entire buffer cache to a single process running AGGRESSIVE. We compared FIXED-HORIZON with AGGRESSIVE, and determined that each algorithm may perform substantially better than the other in particular situations.

Figure 4.3 shows an example situation in which TIP2 leaves a single disk idle and incurs unnecessary stall whereas AGGRESSIVE prefetches as deeply as the disk allows and incurs less stall. In this example, TIP2's assumption of infinite parallelism does not hold and prefetches do not complete in $T_{\text{disk}}$ time. This phenomenon also occurs when there is an ample number of disks, but the accesses are unevenly distributed over the disks. Effectively, only a portion of the disk array is in active use and that portion does not provide enough parallelism to avoid stall.

On the other hand, when there is sufficient disk parallelism, prefetching too aggressively may cause unnecessary disks accesses, each of which adds a CPU overhead of $T_{\text{driver}}$ to the application's execution. Figure 4.4 shows how this phenomenon may occur. Here, TIP2's assumption of infinite parallelism is a reasonable approximation, whereas AGGRESSIVE's assumption that disk accesses have no computational overhead leads to increased application CPU time.

Our study concluded that when stall is anticipated even far in the future, it is better to prefetch aggressively. On the other hand, when there is sufficient parallelism to avoid stall, it is better not to prefetch beyond the prefetch horizon. Based on this intuition, we developed a new algorithm for the SPACE problem called FORESTALL, which prefetches dynamically based on estimates of upcoming load.

## 4.4 FORESTALL

Figure 4.5 is a graphical illustration of a scenario in which upcoming hotspots will cause stall. Disk $a$ represents the ideal case: missing data is well-distributed and there is always enough time to prefetch the data without stalling. TIP2's bounded prefetching works well for this disk. Unfortunately, accesses may come in bursts such as the requests for

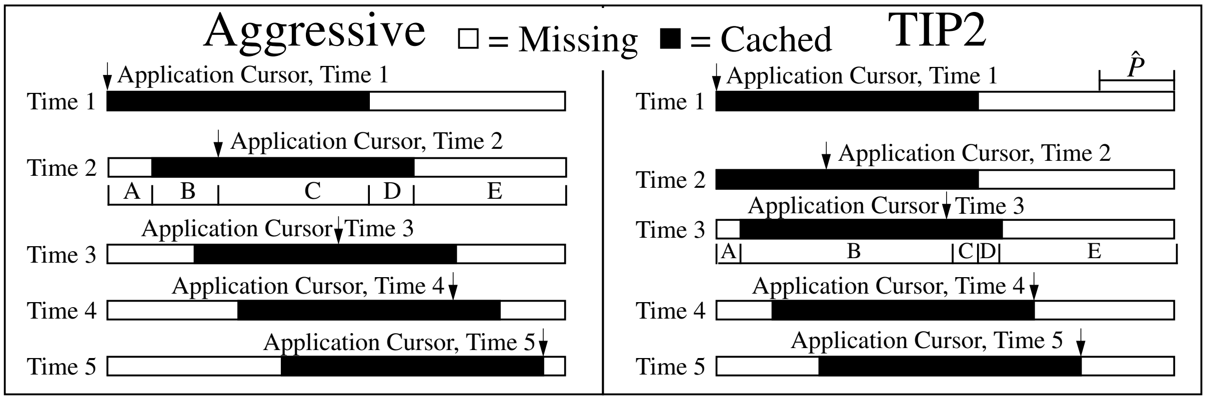



Figure 4.3: Lost opportunities. Initially, an application has a long sequence of future accesses cached, followed by a sequence of missing blocks. The figure shows how TIP2 and AGGRESSIVE proceed, in five snapshots. At time 2, AGGRESSIVE has evicted the blocks in region A in order to prefetch distant data in region D. The blocks in region B have been consumed but there has not yet been time to evict in favor of prefetching missing data from region E. Region C is still cached, and will soon be consumed. AGGRESSIVE keeps the disk busy throughout the sequence. On the other hand, TIP2 does not prefetch when there are no missing blocks within the prefetch horizon, labelled $\hat{P}$ in the figure. At time 3, TIP2 has just begun prefetching and most of the cached data has been consumed. Again, blocks from region A have been evicted to prefetch for region D. The large set of good eviction decisions in region B remain in memory due to conservative prefetching. In general, if a long sequence of accesses is cached, TIP2 allows the disks to go idle even if subsequent prefetching cannot satisfy the later uncached accesses and large stalls result. In contrast, AGGRESSIVE takes advantage of the lull in I/O activity during the read of the cached sequence to prefetch as many missing blocks as possible.

blocks $b1$, $b2$ and $b3$ in the figure. Such a burst necessitates earlier prefetching that TIP2 would fail to perform. Nevertheless, the burst on disk $b$ is small enough that we need not begin prefetching immediately; this disk is not yet *constrained.* Intuitively, a disk is constrained when there is not enough time to prefetch all missing blocks by the time they are needed (the formal definition is given below). On disk $c$, considering only accesses $c1$–$c4$ it appears that there is enough time to prefetch all the blocks. However, because access $c5$ comes so soon after $c4$, it is too late to avoid stalling for $c5$ even if prefetching begins immediately. Access $c5$ constrains disk $c$. The best we can do is to start prefetching now to minimize the stall for $c5$. Note that the access to $c6$ does not change the picture. Since the disk will be fully utilized just to minimize stall for $c5$, there will be no opportunity to prefetch deeply to reduce stall for any subsequent access. Thus, there is no reason to examine a hint sequence beyond a request that constrains the disk. On the other hand, if a disk is nearly-constrained, even a small burst of activity far in

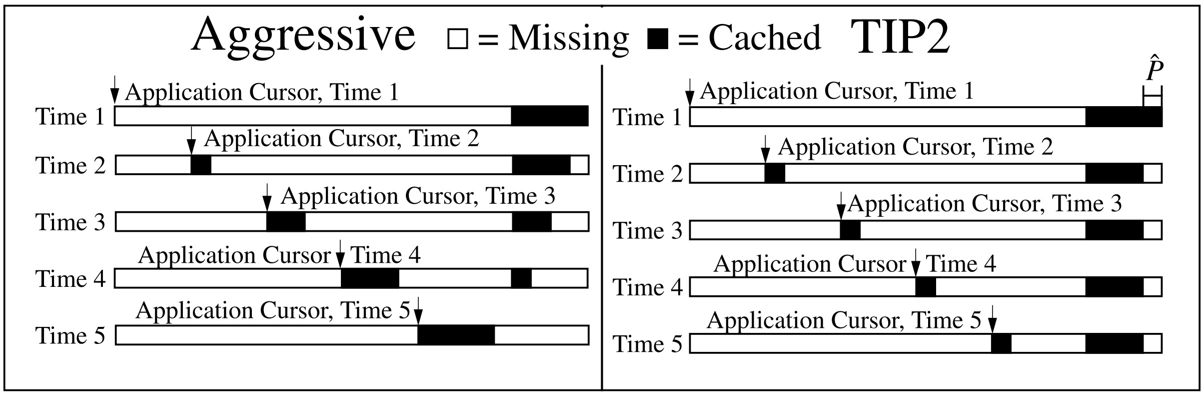



Figure 4.4: Wasted effort. Initially, an application on a system with high I/O bandwidth caches a sequence of blocks in the future, but there are many intervening blocks that must be fetched. The figure shows how TIP2 and AGGRESSIVE proceed, in five snapshots. AGGRESSIVE always ejects a cached block if it can take advantage of an idle disk to prefetch a closer block. As time proceeds, it uses the available bandwidth to prefetch more data, evicting both distant cached data and, when available, just-consumed data. By time 5, AGGRESSIVE has flushed the entire sequence of distant cached data before it can be used. TIP2 prefetches conservatively, and since sufficient bandwidth exists, incurs no additional stall. It leaves the majority of the distant cached sequence in memory, saving the overhead of re-fetching those blocks. Each unnecessary fetch incurs $T_{\text{driver}}$ computational overhead to allocate a buffer, queue the request, and service the interrupt when the request completes; this overhead can be substantial.

the future can constrain the disk and make it necessary to begin the entire prefetching schedule earlier in order to avoid stall.

Informally, FORESTALL sums the time it will take to prefetch all uncached blocks before a given request; if this time is greater then the expected time until the request arrives then the disk is constrained. Failing to initiate deep prefetching on the constrained disk will increase application stall on the request causing the constraint.

Detecting constraint requires estimation of three quantities: the blocks that must be prefetched, the time it will take to prefetch these blocks, and the time it will take the application to consume the intervening cached blocks. FORESTALL determines which blocks must be prefetched based on the following simple cache model: a hinted block will be missing if it is not cached and it is the earliest hint for its block. Essentially, it makes the optimistic assumption that the system will only have to prefetch each missing block once. The estimate of $T_{\text{disk}}$, the time to fetch a block, is the average system I/O time up to that point in the trace. Finally, the estimate of $T_{\text{app}}$, the time between hinted accesses, is the average amount of computation performed by that process between hinted accesses

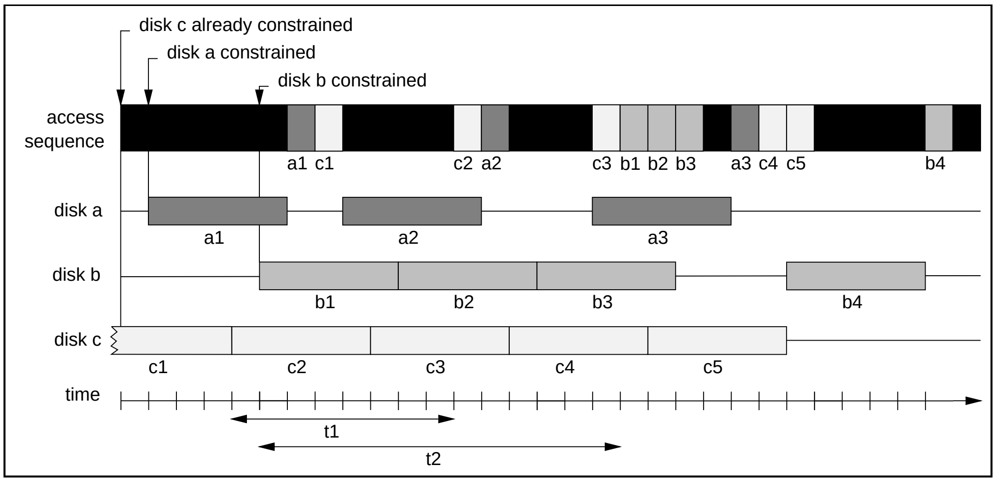


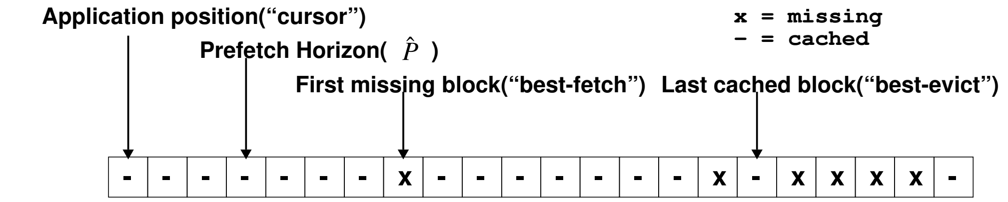

Figure 4.5: Constrained disks. The upper bar represents the future request sequence; black segments represent requests to cached blocks, and all other segments represent requests for missing blocks on one of the three disks. The three lower bars represent schedules for each disk that satisfy all requests in time without incurring stall. Disk $a$ represents the ideal situation because prefetching for each block can begin one fetch time before the block will be consumed. Disk $b$ contains a burst of requests $b1$, $b2$ and $b3$, but we do not need to begin fetching those blocks until we reach the line marked "disk $b$ constrained." The schedule for disk $c$ shows that in order to service all requests without stall, we would have to begin prefetching in the past, and therefore we will incur stall at some point. We say that disk $c$ is currently the only *constrained* disk of the three.

up to that point in the trace.

I now give a formal specification of constrained disks. Let $r_1 r_2 \ldots$ be the sequence of hints. Let $\textsc{Incore}(i)$ be a boolean variable representing our estimate, described above, of whether request $i$ will be in cache or will need to be prefetched. Finally, let $\text{disk}(i)$ be the disk holding request $r_i$. Then, disk $d$ is constrained by request $i$ if and only if $i$ is the smallest positive integer such that $r_i$ is the first reference to an uncached block and

$$\sum_{j \leq i | disk(j)=d} \textsc{Incore}(j) \cdot T_{\text{disk}} > i \cdot T_{\text{app}}. \tag{4.2}$$

This specification of a constrained disk is very similar to that used by the FORESTALL algorithm as presented in our earlier collaborative study [KTP+96], but it differs in two details. First, the INCORE function used here only counts the first access to a block as a miss instead of assuming all accesses to currently uncached blocks will have to be




| Algorithm | Prefetch When |
|---|---|
| FIXED-HORIZON | best-fetch $< \hat{P}$ |
| AGGRESSIVE | best-fetch < best-evict |
| FORESTALL | best-fetch $< \hat{P}$<br>*or*<br>best-fetch < best-evict *and* Disk(best-fetch) constrained |

Figure 4.6: SPACE Algorithms. FIXED-HORIZON is conservative, prefetching only missing blocks within the prefetch horizon. AGGRESSIVE is aggressive, prefetching whenever a more distant block can be evicted. FORESTALL dynamically decides whether to prefetch conservatively or aggressively based on an estimate on upcoming load on each disk.

prefetched. Second, I do not vary the estimate of $T_{\text{disk}}$ in a manner that corresponds to FORESTALL's overestimation $F'$ of the ratio between $T_{\text{disk}}$ and $T_{\text{app}}$.

Even with these simplifying assumptions, identifying a constrained disk could still be very expensive since it could require revisiting every hint every time deep prefetching is considered. In practice, this is not necessary; I will revisit the issue in Section 4.8.

Figure 4.6 summarizes the three SPACE algorithms: FIXED-HORIZON, AGGRESSIVE and FORESTALL.

## 4.5 TIPTOE

TIP2 and LRU-SP/AGGRESSIVE, the two existing systems for informed prefetching and caching, both suffer from the shortcomings of their internal SPACE algorithms. Our joint study showed that these shortcomings arise in practice, and showed that dynamic disk-aware prefetching, as provided by FORESTALL, provides an improved approach to the SPACE problem. This motivates extending both earlier systems to incorporate dynamic prefetching into COST-BENEFIT and LRU-SP. This section considers COST-BENEFIT; the next considers LRU-SP.

Extending COST-BENEFIT requires developing a new set of estimators for the benefit

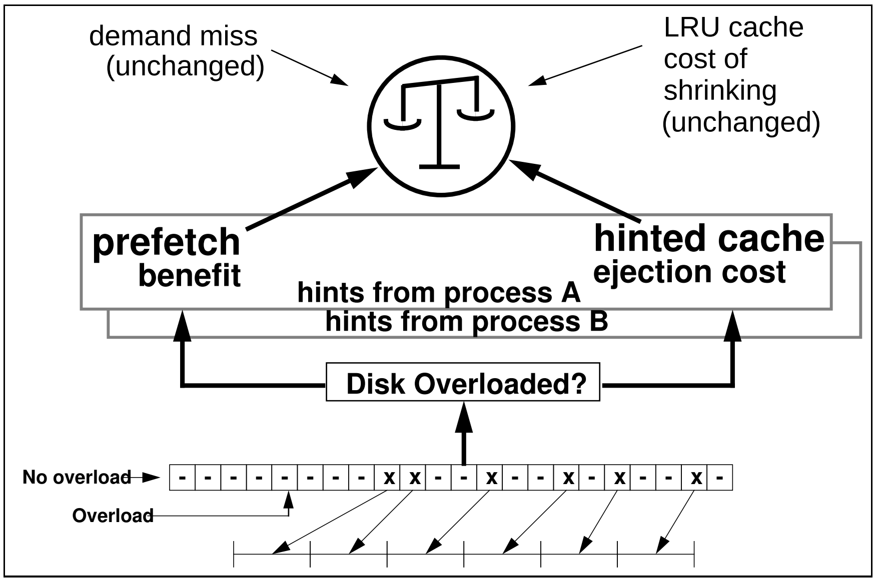



Figure 4.7: The TIPTOE algorithm. Like TIP2, TIPTOE uses cost-benefit analysis to make resource allocation decisions. However, the existing TIP2 estimators of the benefit of prefetching, and the cost of evicting hinted data, are modified based on a calculation of disk load. This calculation depends on the hint sequence, the state of the cache, and rate estimates of application consumption and disk access time. For each disk, TIPTOE determines where in the hint sequence this disk will fall behind the application and cause stall; more heavily loaded disks are then given preference. Details of this calculation are shown in Figure 4.8.

of "deep prefetching" beyond the prefetch horizon, and for the cost of caching hinted data that will not be referenced within the prefetch horizon. These estimators consider disk overload at different points in time, so I will refer to the resulting algorithm as TIPTOE: TIP with Temporal Overload Estimators. Figure 4.7 shows the structure of the TIPTOE algorithm.

### 4.5.1 The Benefit of Deep Prefetching

TIP2's estimator for the benefit of prefetching will never prefetch missing blocks beyond the prefetch horizon. Our joint study showed that in some situations deeper prefetching is necessary; historically, we began working on TIPTOE based on the observation that variations in load, such as long cached sequences followed by "hotspots" of missing data, might require deep prefetching beyond the prefetch horizon. TIPTOE's estimator of the

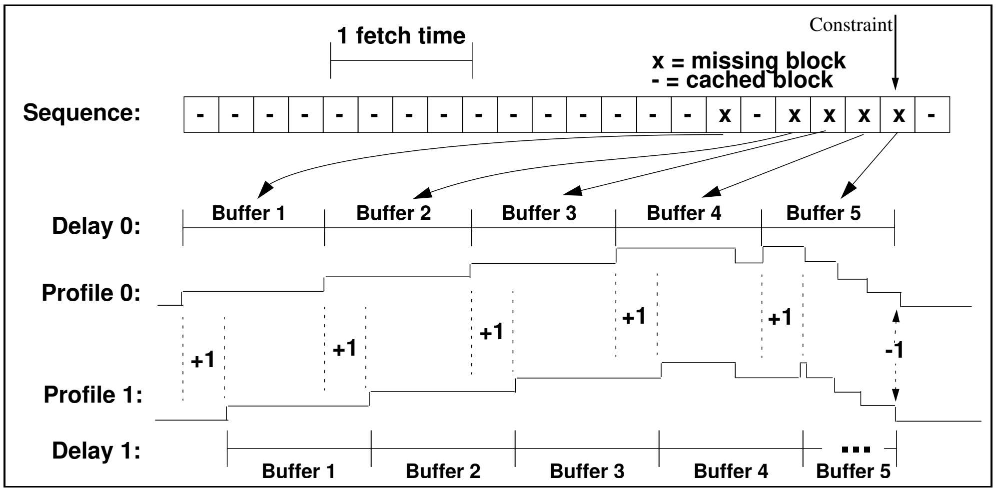



Figure 4.8: The Benefit of Deep Prefetching. For a particular disk, the figure shows a long sequence of cached data followed by a hotspot of missing blocks in the future. The hotspot requires enough I/O that the disk is constrained. The row marked "Delay 0" shows how to schedule fetches if prefetching begins immediately. The next line shows a profile of how many buffers must be dedicated to this sequence of prefetches at each point in time. The profile grows at the beginning because fetches complete but have not been consumed, so the buffers must stay in memory. The next two lines show the schedule and profile that results if prefetching is delayed for one timestep. The difference in total area under the profiles yields the cache resources saved by delaying prefetching for one step — in this case, four buffer-accesses are saved. This saving must be offset against the increase in stall that results from delaying prefetching. As derived in the text, if the disk is constrained $n$ accesses in the future, the marginal change in stall time with respect to buffer usage is $T_{\text{disk}}/n$.

---

benefit of prefetching must be able to identify these situations.

More specifically, if access $r_c$ constrains disk $d$, TIPTOE must determine the benefit of allocating buffers for deep prefetching for that constraint. Expressing this benefit in the common currency requires determining the change in stall time for a given change in buffer usage. Figure 4.8 gives an example that shows the relation between buffer usage and stall time. Delaying deep prefetching one access adds one inter-access period, $T_{\text{app}}$, of stall, but reduces by one access the time each of the deep-prefetch buffers must be held. If the constraining access is $n$ steps in the future then, by the definition of constrained disks, all intervening accesses will be serviced without stall, taking time $nT_{\text{app}}$. This allows $nT_{\text{app}}/T_{\text{disk}}$ disk fetches to complete, so delaying for one access means that each of these fetches will be consumed one access earlier, saving $nT_{\text{app}}/T_{\text{disk}}$ units of bufferage. Then the marginal change in stall with respect to buffer usage is the change in stall ($T_{\text{app}}$) divided by the change in bufferage ($nT_{\text{app}}/T_{\text{disk}}$), which is $T_{\text{app}}/(nT_{\text{app}}/T_{\text{disk}}) = T_{\text{disk}}/n$. The complete common-currency benefit of prefetching is then:[3]

$$\text{Benefit}(\text{prefetch}, x) = \begin{cases} T_{\text{disk}} & x = 0 \\ \frac{T_{\text{disk}}}{x(x+1)} & x < \hat{P} \\ 0 & x \geq \hat{P}, \text{disk}(x) \text{ unconstrained} \\ \frac{T_{\text{disk}}}{x} & x \geq \hat{P}, \text{disk}(x) \text{ constrained} \end{cases}. \tag{4.3}$$

### 4.5.2 The Cost of Ejecting from a Constrained Disk

Ejecting a hinted block requires that the block be prefetched back at a later time. The change in I/O service time is $T_{\text{driver}}$ to prefetch the block back plus any stall that will be incurred on the eventual access to the ejected block. On a constrained disk, there is no opportunity to prefetch in advance and mask the latency of the access. Thus, the access will add a full $T_{\text{disk}}$ of stall to the application's I/O service time. If the block will be read in $x$ accesses then ejecting it will free the buffer until the block must be read back in, $x$ accesses later. In terms of the common currency, the ejection causes $T_{\text{disk}}$ stall in exchange for $x$ units of bufferage: a cost of $T_{\text{disk}}/x$. The final estimator for the cost of ejecting a block from the hinted cache, incorporating constrained disks, is:

$$\text{Cost}(\text{Hinted\_eviction}, y) = \begin{cases} T_{\text{driver}} + T_{\text{disk}} & y = 1 \\ T_{\text{driver}} + \frac{T_{\text{disk}}}{y-1} & 1 < y \leq \hat{P} \\ \frac{T_{\text{driver}}}{y-\hat{P}} & y > \hat{P}, \text{disk}(y) \text{ unconstrained} \\ \frac{T_{\text{driver}}+T_{\text{disk}}}{y} & y > \hat{P}, \text{disk}(y) \text{ constrained} \end{cases}. \tag{4.4}$$

These new estimators have two useful properties. The benefit of prefetching on a constrained disk with constraint at distance $x$ is $T_{\text{disk}}/x$, while the cost of evicting a block at distance $x$ from a constrained disk is $(T_{\text{disk}} + T_{\text{driver}})/x$. First, note that $T_{\text{disk}} \gg T_{\text{driver}}$, so these values are similar. So it is worth initiating prefetching if blocks can be found that are more distant than the constraint being fetched for; if, however, the only available blocks will be used before the constraint then the cost of evicting these blocks will be too high to justify prefetching. This matches the intuition that an algorithm should not throw out early blocks to prefetch for later blocks, if the blocks all lie on constrained

[3]Note that the discontinuity at $x = \hat{P}$ occurs because TIP2's benefit estimator assumes infinite parallelism, and therefore amortizes the benefits of a deeper prefetch over many disks. TIPTOE's model is aware of disk layout, but does not model the interaction between fetches on different disks, and therefore computes the benefit for each disk as if that disk were the primary constraint.

disks (the do-no-harm rule). Second, the benefit of prefetching is smaller than the cost of eviction by a $T_{\text{driver}}/x$ term. This difference adds some hysteresis — the system is more likely to keep a block in the cache than it is to load the block in the first place, reducing the likelihood of thrashing the same block in and out of the cache.

## 4.6 LRU-SP/FORESTALL

TIPTOE represents the incorporation of FORESTALL, the best known SPACE algorithm, into cost-benefit allocation. In order to provide a meaningful comparison of allocation algorithms, I also incorporate FORESTALL into LRU-SP. The modular nature of LRU-SP makes it straightforward to "plug in" FORESTALL in place of AGGRESSIVE.

This results in a set of four algorithms: TIPTOE and LRU-SP/FORESTALL, representing the best known SPACE algorithm incorporated into two radically different approaches to allocation, and their predecessors, TIP2 and LRU-SP/AGGRESSIVE. All experiments are performed for all four of these systems.

## 4.7 A Study of Embedded Allocation Algorithms

I have described two allocation algorithms: COST-BENEFIT and LRU-SP. This section gives some intuition for how each operates, and for how they should compare.

### 4.7.1 Synthetic Workloads and Caching

I will demonstrate through a micro-benchmark that LRU-based allocation may give buffers to applications that consume data quickly but have limited re-use, rather than to applications that consume more slowly but derive higher overall benefit from buffers because they have higher re-use. As mentioned above, to first approximation LRU-SP allocates buffers to processes according to their access rate (see Section 4.7.2 for details); thus, LRU-SP will cache data for an application that has a high data consumption rate but low re-use. I consider the XDS trace described above running in parallel with a synthetic process that repeatedly reads a 500-block file sequentially. This artificial process computes for 10 ms between each read; XDS computes for 700 microseconds between reads on average. Figure 4.9 shows the results of varying cache size from 500 blocks to 1500 blocks.

The non-hinting process uses LRU replacement, so it attains no benefit from fewer than 500 buffers. With 499 buffers, for instance, each block is evicted immediately before

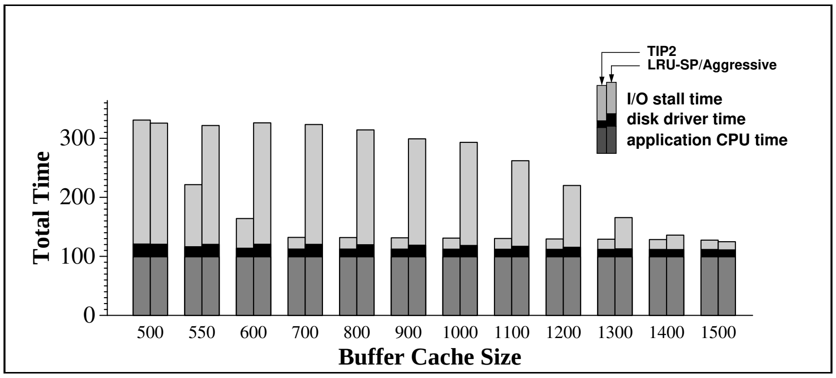



Figure 4.9: XDS takes buffers from a synthetic process with high re-use. The synthetic process loops through 500 blocks without hints, so with fewer than 500 cache buffers it achieves no re-use. COST-BENEFIT notices that XDS does not need buffers, while the synthetic process has high re-use, and so dedicates most of the cache to the synthetic process. LRU-SP allocates according to rate, and requires a cache of around 1300 buffers before it dedicates 500 buffers to the synthetic process.

it will be read. With 500 cache buffers neither algorithm derives any caching benefit because some blocks are used to perform I/O's for XDS so the synthetic process never gets a full 500 buffers. With 550 cache buffers, however, TIP2's cost-benefit estimators conclude that caching data for the non-hinting process is more important than prefetching ahead for the hinting process. LRU-SP splits the buffer cache according to the relative rates of the processes, giving much of the cache to the hinting process even though it does not re-use its data. The high rate of XDS relative to the non-hinting process means that the cache must become quite large (around 1300 buffers) before LRU-SP will allocate sufficient buffers to hold the 500 buffer working set.

### 4.7.2 LRU-SP and Rate-Based Allocation

In this section I give a more concrete analysis of the allocation resulting from LRU-SP. I will show that allocation is rate-based, but also dependent on the re-use characteristics of the processes. Finally, I will show that, given information about re-use, it is possible to compute exactly the resulting steady-state allocation.

First, consider two processes $A$ and $B$ running alongside, each of which has no re-use. Let their relative consumption rates be $r_A$ and $r_B$, so the first process reads $r_A$ pages every time the second process reads $r_B$ pages. For concreteness, let $r_A = 20$ and $r_B = 10$. Then process $A$ is placing 20 blocks into the LRU queue for every 10 of process $B$. We expect that process $A$ will own 20 out of each 30 blocks in the queue, for a total of 2/3 of the buffers.

Instead, consider the opposite situation in which process $A$ reads at rate $r_A$ again, but always touches the same block, while process $B$ again has no re-use. In this case, the rates are not relevant: process $A$ will own a single block of the cache, and process $B$ will own the rest. So the eventual allocation depends on both rate and re-use.

Let the cache have size $n$, and let the *access profile* of process $A$, $(a_1, a_2, \ldots, a_n, a_{n+1})$, be a distribution showing what fraction of $A$'s requests go to the first element of the cache $(a_1)$, the second element $(a_2)$, the last element $(a_n)$, and finally what fraction are cache misses $(a_{n+1})$. So $\sum_{i=1}^{n+1} a_i = 1$. Similarly for $b_i$.

I assume that each request is chosen independently from the access profile of process $A$ with probability $p = r_A/(r_A + r_B)$, and independently from the access profile of process $B$ with probability $1 - p$. Let $p_i$ be the probability that process $A$ owns the $i^{th}$ element of the LRU. Then $p_1 = p$, since at each round process $A$ will own the first element with probability $p$, and process $B$ will own the first element with probability $1 - p$.[4] For convenience, I define the following cumulative probabilities for $2 \leq i \leq n$:

$$
\begin{aligned}
c_i^A &= \sum_{j=1}^{i-1} a_j \\
c_i^B &= \sum_{j=1}^{i-1} b_j
\end{aligned}
$$

.

Assume a vector $(p_1, p_2, \ldots, p_n)$ gives the $p_i$ values during a particular round of execution. The recurrence for $(p'_1, p'_2, \ldots, p'_n)$ , the vector during the next round, is:

$$
\begin{aligned}
p'_1 &= p_1 \\
p'_i &= p_{i-1}\left(p(1 - c_i^A) + (1-p)(1 - c_i^B)\right) + p_i\left(pc_i^A + (1-p)c_i^B\right), 2 \leq i \leq n.
\end{aligned}
$$

The last equation requires some explanation. When a request arrives, one of two things can happen to $p_i$: first, the request could be for an earlier element of the LRU, in which case $p_i$ would remain unchanged. Second, the request could be for $p_i$ or a later element (or a cache miss), in which case the former $i - 1^{st}$ element would become the $i^{th}$ element, so $p'_i$ will be $p_{i-1}$. The request will be for an element before $i$ with probability $pc_i^A + (1-p)c_i^B$ because with probability $p$, process $A$ will make the request,

[4] This assumes that the two processes share data and pass blocks back and forth; an alternate approach that would introduce additional complexity would be to assume that the two processes access distinct pools of data.

and $c_i^A$ is defined as the cumulative probability that a request from process $A$ occurs in $\{1 \ldots i-1\}$. Likewise, with probability $1-p$, process $B$ makes the request and $c_i^B$ is the appropriate cumulative probability. For convenience, let $x_i = pc_i^A + (1-p)c_i^B$ and $y_i = p(1-c_i^A) + (1-p)(1-c_i^B)$, so the recurrence becomes:

$$\begin{aligned} p'_1 &= p_1 \\ p'_i &= p_i x_i + p_{i-1} y_i, 2 \le i \le n. \end{aligned}$$

In matrix form:

$$\begin{pmatrix} p'_1 \\ p'_2 \\ p'_3 \\ \vdots \\ p'_n \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 & 0 \\ y_2 & x_2 & 0 & \cdots & 0 & 0 \\ 0 & y_3 & x_3 & \cdots & 0 & 0 \\ & \vdots & & \ddots & & \\ & & & & y_n & x_n \end{pmatrix} \begin{pmatrix} p_1 \\ p_2 \\ p_3 \\ \vdots \\ p_n \end{pmatrix} \tag{4.5}$$

Solving this equation for the steady-state probabilities requires finding the eigenvector corresponding to the eigenvalue 1; since the matrix is triangular, this is straightforward, and yields:

$$p_i = p \prod_{j=2}^{i} \frac{y_j}{1-x_j}, 1 \le i \le n. \tag{4.6}$$

## 4.8 Implementation Details

The descriptions above convey the central issues in implementing each of the four systems. But there are many details to the implementation; I describe those details here so the results will be reproducible.

### 4.8.1 Hint Tracking

In the presence of missing and incorrect hints, a tracking algorithm maintains alignment between the request sequence and the hint sequence. All four algorithms use the same tracking system. The system works as follows. Tracking is performed whenever a request arrives. If the request matches the head of the hint sequence, the head is considered a match, and removed. Otherwise, the tracker looks for the request in a fixed-size window of the hint sequence, starting one hint past the head. In our implementation, this window has size 2. If the request is found within this window, all hints up to the first match are dropped, the first match is considered to be the correct match, and is dropped to

```
track_hints(request)
{
        count = 0; found_match = FALSE;
        next_hint = hints->head;
        while (count++ <= MAX_HINT_DELAY) {
                if (request matches next_hint) {
                        found_match = TRUE;
                        break;
                }
                next_hint = next_hint->next;
        }

        if (found_match) {
                drop count hints;
        }
        return;
}
```

Figure 4.10: Hint Tracking Algorithm. If the earliest match of a request in the hint stream falls within a fixed window of the beginning of the hint stream, that match is considered valid. The matching hint and any earlier hints are dropped. Otherwise the request is assumed to be unhinted.

match the request. If the request is not found within the window, it is assumed to be an unhinted request and no hints are dropped. Pseudo-code for this tracking procedure is shown in Figure 4.10.

If a spurious match is found then up to three valid hints will be dropped. But, unless a strange string of coincidences occurs, the next three hinted reads will simply be considered to be unhinted requests, and subsequently the tracking will be back on sequence. This trivial algorithm maintained tracking properly in all cases for our benchmarks. In the future, it will be necessary to improve this algorithm, especially in the presence of automatically-generated hints that may be far less accurate than our programmer-generated hints. In particular, the current algorithm will never regain tracking if `MAX_HINT_DELAY` bad hints arrive in a row.

### 4.8.2 LRU Profiling

TIP2 and TIPTOE require an estimate of the hit rate of the LRU cache. I perform this profiling using the same "segmenting" technique as Patterson *et al.*, described in [PGG+95]. However we implement aging differently. Every 1000 requests Patterson *et al.* recompute a new LRU hit rate profile, and compute a weighted average of this new profile with the existing profile. Thus, the profile at each point is simply a sum of all existing profiles up to that point, weighted by a decaying exponential. My implementation does not perform aging, and simply maintains the overall hit-rate profile of the simulation so far.

### 4.8.3 Epochs

TIPTOE must know which disks are constrained by which requests whenever it decides whether or not to prefetch. To avoid recalculating these constraints from scratch at each access, the implementation breaks the hint sequence into sequential pieces called "epochs." Each epoch keeps track of the number of accesses and missing blocks within the epoch. The contribution of the entire epoch of hints to Equation 4.2 can then be calculated in a single step. Whenever a block is loaded or evicted, we perform a single constant-time operation to find the earliest hint for that block, look up the associated epoch, and modify the count of missing blocks within that epoch. To detect a constraint, it is only necessary to sum the contributions of the epochs and not the hints individually. Epochs reduce algorithmic overhead by a large constant factor. In our implementation, the target size for epochs is 100 accesses, though the algorithm admits other implementations that place distant hints into larger epochs.

The epoch calculation will correctly determine whether epoch endpoints constrain the disks. However, for large epochs, it is possible that an initial sequence of missing blocks could constrain the disk but that later cached data would result in the final endpoint being unconstrained. The application would stall early in the epoch then allow the disk to idle later. This error is especially dangerous for early epochs, as the allocator is likely to prefetch for early constraints. I address this issue by examining each hint in the first few epochs. Under this scheme, assume that epochs have size $n$ and there are $h$ hints, for $h/n$ epochs. If TIPTOE looks through the first $k$ epochs hint-by-hint, the total cost for calculating constraints will be $O(kn)$ to look through the first $k$, plus $O(h/n)$ to look at each epoch. Minimizing $kn + h/n$ with respect to $n$ gives $k - hn^{-2} = 0$ or $n = \sqrt{h/k}$. Thus, if TIPTOE examines the first two epochs, its epoch size should be the square root of half the number of pending hints.

### 4.8.4 When to Consider Prefetching

Because deep prefetching is only relevant over large numbers of accesses, it is not necessary to compute the deep prefetching constraints at every access. Instead, we further reduce overhead by recomputing only every 5 accesses.

With or without epochs, the computational overhead is bounded, not by the number of hints, but by the benefit of prefetching. Because the benefit of prefetching from a constrained disk falls off linearly with the distance to the constraint, at some point the benefit will be smaller than the cost of the lowest cost buffer. There is no need ever to examine the hints beyond this point. The exact point at which this happens depends dynamically on the current cost of taking a buffer from a cache supplier.

TIP2 considers issuing prefetches whenever a change in cost or benefit calculations creates the chance that there might now be sufficient benefit from prefetching to merit a buffer. Most often, this occurs when the application consumes a hinted block which shifts all remaining hints one access closer. In particular, it shifts a hint from beyond the prefetch horizon to within the prefetch horizon which raises the benefit of prefetching from zero to some positive value. In contrast, deep prefetching according to LRU-SP/AGGRESSIVE, TIPTOE or LRU-SP/FORESTALL must be disk-aware because it takes advantage of disk idleness to prefetch more deeply regardless of application activity. These algorithms must also consider prefetching whenever a disk goes idle. For consistency, the simulator adopts the policy that all algorithms consider prefetching at each request, and whenever a disk goes idle, but that benefits of deep prefetching are recalculated every 5 requests.

### 4.8.5 Prefetching on Multiple Idle Disks

If multiple disks are idle, there may be limited resources that must be allocated among the idle disks. For TIP2, this is not an issue since first, each process will only dedicate up to $\hat{P}$ blocks to prefetching, and second, the benefit of prefetching for requests within the prefetch horizon is sufficiently high that the allocator will not deny buffers to a process wishing to prefetch. The other algorithms proceed as follows: in unspecified order, each process is given an opportunity to prefetch. The active process considers each hint in temporal order, and decides whether to prefetch that block according to the current prefetching algorithm.

### 4.8.6 Writes and Dirty Bits

The writes in the traces are all delayed writes — they set a dirty bit rather than beginning a synchronous I/O. However, the existing TIP2 and LRU-SP/AGGRESSIVE systems deal with dirty bits in different ways. In TIP2, no dirty block is written back until that block is both the lowest-cost block, and lower cost than the highest-benefit I/O. At that point the dirty block is written, and bidding begins again. LRU-SP/AGGRESSIVE does not explicitly suggest a policy, so I implement the following: whenever a block is about to be evicted, its dirty bit is checked. If the block is dirty, it is written asynchronously, and a different block is found and returned.

However in all the primary experiments described below, delayed writing is turned off since the writeback policy has impact on the overall execution time.

### 4.8.7 Disk Queueing

Should the prefetcher issue just one prefetch at a time per disk, or should it issue more? Modern disks and their low-level device drivers are capable of reordering fetches to reduce average disk service time and increase effective disk bandwidth. A prefetcher can exploit this capability by queuing multiple prefetches at the device. In general, longer queues provide greater reordering opportunities and larger reductions in average disk service time. Moreover, the same positioning effects that make disk fetch reordering effective suggest that cache evictions should be sensitive to disk location. For many workloads, especially sequential ones, proximity in the hint sequence is a good indicator of proximity on disk. In such cases, simultaneously evicting larger numbers of neighboring blocks in the hint sequence can reduce disk service time for the re-fetches of the evicted blocks.

A number of costs offset these benefits of simultaneously issuing large numbers of prefetches. First, large queues tie up cache buffers that might be better used to cache data for re-use. Second, filling a large queue forces earlier replacement decisions which may itself reduce cache effectiveness. Lastly, very deep queues allow early prefetches to be reordered behind many other, later prefetches which may lead to unnecessary stall.

Addressing this issue at a theoretical level requires extending the system model to non-constant disk times, and incorporating disk geometry and layout. This is a daunting task. Modifying the depth, order, or nature, of a prefetching schedule in a general, extensible manner based on this kind of information seems to be an extremely difficult problem. I do not address this issue in full. Instead, since the advantages of sorting are dramatic, I take the approach taken by other researchers of incorporating an independent *a priori* scheme to allow multiple simultaneous prefetches to be sent to the disks at once, without reasoning on-the-fly about the explicit benefits of doing so.

LRU-SP/AGGRESSIVE, TIPTOE and LRU-SP/FORESTALL issue requests in groups of up to 16, a technique that Cao has called batching [CFKL95b]. In my implementation the technique works as follows. Prefetching is only initiated on idle disks. When a system decides to prefetch on a particular disk, it continues to submit I/O's until the batch is full or the algorithm no longer chooses to prefetch. At that point, the prefetcher issues the batch of requests, and the disk is no longer considered idle. The prefetcher continues to try to fill batches for the remaining idle disks.

TIP2, on the other hand, never prefetches more then $\hat{P}$ accesses ahead into the request stream, so it is never in danger of dedicating too many buffers to prefetching. Thus, TIP2 performs no batching, and tends to have somewhat deeper queues than the other algorithms when there is little re-use and the application is I/O bound.

### 4.8.8 Multiple Estimators

In TIP2 and TIPTOE it is possible for multiple estimators to value the same block. For instance, if a recently-read block also has a future hint, the LRU estimator and the hinted cache estimator will both publish a cost of evicting the block. In this case, the value of the block is taken to be the maximum of the values bid by the estimators. Imagine that two processes will use the same block in the future, but the first one will use it earlier. By the time the second one needs the block, even if it was evicted, it will already have been re-read by the first process. This observation motivates the use of the maximum estimator rather than, for instance, the sum.

### 4.8.9 Multiple Trackers

A similar problem arises for LRU-SP/AGGRESSIVE and LRU-SP/FORESTALL. Say processes $A$ and $B$ track (have hints for) the same block $x$. Process $A$ is asked to give up a block and chooses the cached block which it will reference latest, which happens to be $x$. However, process $B$ also tracks block $x$ and will use it earlier. Whenever this situation arises, I simply evict the globally least-recently-used block.

### 4.8.10 Posthint Estimation

The next implementation issue is specific to TIP2 and TIPTOE. When a hinted read arrives and there is no future hint for the same data, the system must decide how long the block should be kept in memory. In the original TIP2 system the block was added to the tail of the LRU queue under the assumption that, since it was recently accessed, it might be accessed again in the near future. However, if an XDS-like process is streaming through a large amount of hinted data with minimal re-use, and another process like POSTGRES2 is performing unhinted reads with strong locality[5] this policy will dilute the LRU queue with buffers that are never re-used (this problem is often referred to as "pollution" of the buffer cache). The opposite policy is to take lack of hints for a block as a "release" of the block, and place the block on the head of the LRU queue for immediate eviction. However under this policy a process such as SPHINX, which offers hints in small

[5] The same situation could arise within a single process.

batches just before the data is required, would be prone to flush blocks that might soon be hinted.

The cost-benefit framework provides a simple, elegant solution to this problem. Rather than inserting these problematic "posthint" buffers into the LRU queue, the system instead inserts them into a separate posthint queue which maintains an independent estimate of the value of its buffers. If the posthint buffers are often re-read, as in SPHINX's case, the allocator will choose to grow the posthint cache at the expense of the LRU cache. On the other hand if the posthint buffers are never accessed but unhinted accesses demonstrate re-use, as for POSTGRES2 and XDS, the allocator will instead choose to dedicate resources to the LRU cache.

In general, the cost-benefit framework allows the system designer to identify subclasses of a resource that display uniform or similar patterns of behavior or re-use. The designer can then tailor estimators to each subclass, such as posthint buffers or unhinted buffers. Buffers can be members of multiple classes, and can be valued by multiple estimators,[6] and the allocator will automatically incorporate any new estimates into the global valuation described earlier.

### 4.8.11 Dynamic Parameter Estimation

TIPTOE requires a dynamic estimate of $T_{\mathrm{app}}$ and $T_{\mathrm{disk}}$. $T_{\mathrm{app}}$ is estimated at each point of the trace to be the average inter-access process computation time up to that point. $T_{\mathrm{disk}}$ is estimated as the average media access time up to that point in the trace.

There are several effects that are not caught by these estimates. First, disk time is taken as an average over all processes rather than a per-process average. Disk times within our applications do not tend to vary much, but in the worst case a sequential process might attain an average I/O time under 4 ms, while a process performing random accesses might pay closer to 15 ms on average.

Next, there are effects that depend on the intended use of the estimators. TIPTOE uses the $T_{\mathrm{app}}$ estimator to determine how long it will take an application to consume a cached sequence of data blocks, but the estimate as given is not valid if other processes are sharing the CPU. TIPTOE could instead track the fraction of CPU usage enjoyed by each process, and scale the computation time by the reciprocal of this factor, but this alone is not sufficient because the CPU might not be saturated. If this process uses the CPU 10% of the time, other processes use the CPU for another 10%, and the CPU is

[6]For instance, a block could be read as a demand miss, placed into the LRU queue, and then have a hint arrive that says it will be read again in 300 accesses. The LRU estimator and the hinted cache estimator will independently value the buffer, and if either estimator assigns high cost the buffer will not be evicted.

idle for the remaining 80%, then computation during consumption of cached data can grow to use the large chunk of idle time. These two observations suggest that a more correct estimate of process $A$'s CPU time requires two parameters: the fraction of time the CPU is idle ($r_1$), and the fraction of the remaining time that the CPU is being used by process $A$ ($r_2$). Given these parameters and the prior estimate of process compute time $T_{\text{app}}^{\text{old}}(A)$, the new estimate is

$$T_{\text{app}}(A) = T_{\text{app}}^{\text{old}}(A) \cdot (r_1 + (1 - r_1)r_2) .$$

Similarly, the estimate of disk time does not take into account disk contention between processes. If process $A$ gets the disk for half of each time unit on average and other processes split the other half, then the expected time to load a block should be twice the access time. However, again, if the disk is idle for some fraction of time, then the idle time could be given over to process $A$ in entirety. Thus, if $d_1$ is the fraction of the total disk use time that the disk is being used by process $A$,[7] and $d_2$ is the fraction of total time that the disk is in use, and $T_{\text{disk}}^{\text{old}}(A)$ is the average access time for process $A$, then the new time should be given by

$$T_{\text{disk}}(A) = T_{\text{disk}}^{\text{old}}(A) \cdot (d_1 + (1 - d_1)d_2) .$$

The existing estimates, though much simpler, seem to perform well, so I have not implemented these more complex estimators. In the presence of larger variances and larger multiprogramming loads, the new estimators might be necessary.

### 4.8.12 Thrashing

During prototyping of these four systems I often found poor performance due to thrashing. At a high level, there are two types of thrashing. First, over-aggressive prefetching could result in undervaluation of caching, causing blocks that would eventually be re-used to be evicted in favor of data blocks that will be used earlier. If the prefetching could be delayed without introducing extra stall, some of the blocks available for re-use could be kept in cache, avoiding the driver overhead of submitting I/O's to re-read the blocks. In this case, every prefetched block is read. Forestall was designed to understand and control this type of thrashing.

This subsection is concerned with the second type of thrashing, in which blocks are prefetched then evicted *before* they are read. This could occur because one process evicts the prefetched blocks of another process, or because a single process evicts blocks fetched on one disk during one batch to fetch data needed sooner on another disk.

I adopt the following prefetching control mechanisms:

[7]Note that this fraction should be computed by time, not number of accesses.

1. Prefetches that will cause another prefetched but not consumed block to be evicted are not allowed. In the presence of incorrect hints this restriction would have to be relaxed.

2. Prefetching is only allowed to fill a certain fraction of the cache with data, in our case 2/3. In TIP2 this limit is not an issue since each process only prefetches up to 68 blocks, but for the other algorithms the limit may be attained. In particular, with large arrays LRU-SP/AGGRESSIVE will attempt to read as deeply as possible into the hint sequence.

3. The $k$ most-recently-read blocks may not be evicted; currently the value of $k$ is 3. This restriction is necessary because occasionally a single hint will be given for a block which will then be read with several system calls. If there is no future use for the block it will be placed into the posthint queue and possibly evicted immediately if the posthint queue has little value.

# Chapter 5

# Single-Process Informed Prefetching and Caching

*Cache, and cache again, deep in the ground and sea,*
*and where it is neither ground or sea.*

*— Walt Whitman, "Leaves of Grass"*

This chapter compares the algorithms of Chapter 4 in the single-process domain. This work began as a joint study [KTP+96] between myself, Hugo Patterson and Garth Gibson at the Parallel Data Lab at CMU; Tracy Kimbrel, Anna Karlin and Brian Bershad at the University of Washington; Pei Cao at the University of Wisconsin; and Kai Li and Ed Felten at Princeton University.

Section 5.1 describes the differences between our joint study and the work below. Then Section 5.2 presents and describes the performance of each algorithm on each of the traces in Chapter 3. Section 5.3 examines some of the issues raised by these results in greater detail. Finally, Section 5.4 gives a distillation of the lessons learned in the single-process case.

## 5.1 Evolution of a Joint Study

Most importantly, the joint study mentioned above reached a set of conclusions which are echoed in the results below. However, the traces used for the study did not perform accurate modeling of unhinted accesses and hints that "trickle in" over time, while the environment described in Chapter 3 captures these effects. All the results below are taken in the new environment, so the numbers and in some cases the applications themselves have changed. Also, the joint study included another algorithm called REVERSE-AGGRESSIVE which I do not simulate (a brief description of REVERSE-AGGRESSIVE, and the reasoning behind the decision not to simulate it, appear in Section 4.2.2).

The difference in the traces from the joint study to the current work reflects a difference in the question being asked. Our joint study chose to focus on algorithms for the SPACE (Standalone Prefetching And Cache Eviction) problem, as defined by the following model: A single application runs alone on a dedicated system with full knowledge of all future requests. The focus of this chapter is slightly different. I study algorithms for informed prefetching and caching in the single-process case. This often corresponds to the SPACE problem, but the presence of unhinted accesses means that some buffers must be dedicated to caching for an unhinted stream. Similarly, the presence of late-arriving hints means that hinted re-use may not be apparent until a needed block has already been evicted.[1]

In this chapter I will discuss and demonstrate the full range of SPACE algorithm issues that we uncovered in our joint study. I will also note situations in which the abstraction of complete foreknowledge does not hold, and will discuss the issues that arise.

## 5.2 Performance by Application

There are two caveats to the numbers appearing below. First, these numbers represent the execution times of a reasonably careful implementation of each algorithm — not highly tuned, but coded with some attention to the impact of details such as the estimator for consumption rate of an application, the method for breaking ties when multiple disks are available for prefetching, and so on. In some cases, a poor implementation may visibly alter the behavior of the algorithm; in these cases I include a detailed description to make it clear exactly how the algorithm achieves its performance.

Second, these algorithms are complex and the graphs show only a small number of statistics (the graphs are stacked bar charts showing total execution time broken into algorithm computation, disk driver computation, and stall). There are interesting effects below the surface when, for instance, two algorithms attain the same overall execution time in different manners — especially for traces that have high-level structure such as DAVIDSON or POSTGRES1. The graphs include visible examples of all known primary effects in the single process case, but I also try to describe the behavior "under the hood" to make it clear how the algorithms differ. When these issues are more global than a single trace, I defer the discussion to Section 5.3, and give forward pointers.

[1] For strict SPACE problems, TIPTOE and LRU-SP/FORESTALL would perform identically. In the traces below, due to unhinted requests and late-arriving hints, they do not.

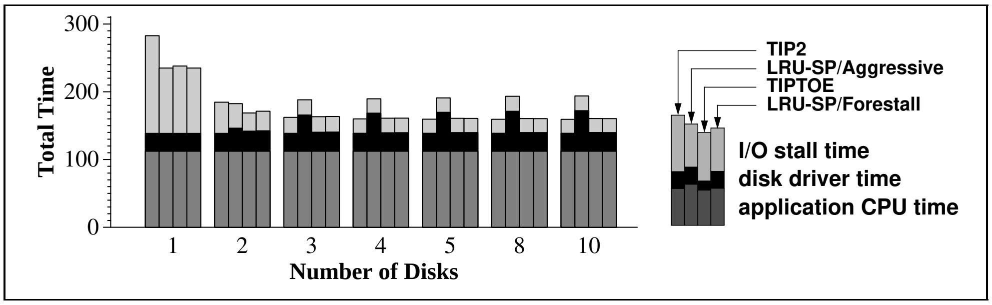



Figure 5.1: Standalone DAVIDSON, four prefetching and cache management algorithms.

### 5.2.1 High Re-Use: DAVIDSON

DAVIDSON has substantially more re-use than any other application in the suite. Recall from Section 3.4.1 that the access pattern loops over 2089 blocks 51 times. On a single disk, there is one significant effect. As described in Figure 4.3, TIP2 sometimes fails to take advantage of transient disk idleness to perform deep prefetching, resulting in a 19% increase in overall execution time. TIPTOE, LRU-SP/AGGRESSIVE and LRU-SP/FORESTALL all perform similarly; in fact, LRU-SP/AGGRESSIVE and LRU-SP/FORESTALL perform identically, showing that as expected the single disk is always constrained, and TIPTOE's overall execution time is within half a percent.

On larger arrays, LRU-SP/AGGRESSIVE prefetches too aggressively for DAVIDSON, and achieves little re-use; this effect is described in detail in Figure 4.4. This leads to unnecessary prefetches which add substantial CPU overhead to the elapsed time. As expected, none of the other algorithms suffers from this problem. More specifically, driver time for TIP2 remains consistent at about 26 seconds, and for LRU-SP/AGGRESSIVE on 2 disks is 34 seconds, on 3 disks is 53 seconds, and on 10 disks is 60 seconds.

These two effects, under-prefetching by TIP2 on one or two disks and over-prefetching by LRU-SP/AGGRESSIVE on larger arrays, are the primary effects. There is one other small effect. As the array gets larger, LRU-SP/AGGRESSIVE, TIPTOE and LRU-SP/FORESTALL all prefetch more than TIP2 and incur a slight increase in driver time. On 2 disks, when tip is still under-prefetching, the FORESTALL-based algorithms incur 14% additional driver cost. On 3 disks this drops to 7%, to about 4% out to 10 disks.

Section 5.3 gives more details about the nature of looping traces, and the influence of batching and prefetching policies on the cache state of an algorithm serving a looping sequence.

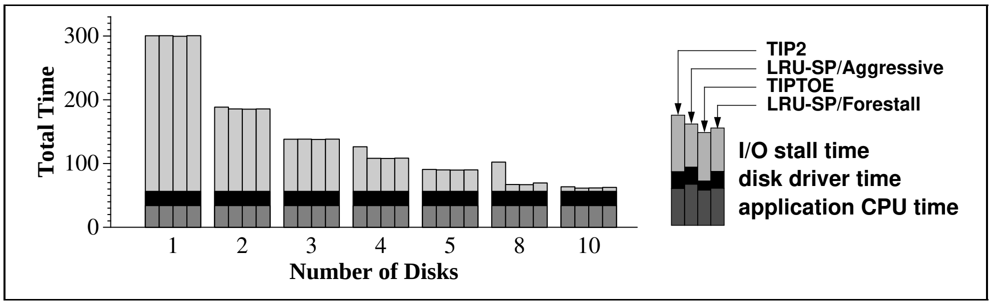



Figure 5.2: Standalone XDS, four prefetching and cache management algorithms.

### 5.2.2 Unbalanced Accesses: XDS

XDS consumes data quickly with little re-use, and displays a strongly unbalanced access pattern across array sizes that are powers of 2. On a four-disk array, for instance, there are stretches of hundreds of accesses that only touch two disks. TIP2 does not initiate prefetching for data more than $\hat{P}$ accesses away, so for long stretches it leaves two disks idle on a four-disk array. When the situation changes, and many reads must be serviced from the two previously idle disks, those two disks cannot provide sufficient bandwidth to keep up with the request stream. It is necessary to begin prefetching early on these disks to service the hotspot without stall when it arrives. Figure 4.5 shows how FORESTALL is designed to recognize and remedy this situation.

Deep prefetching in TIPTOE and in both of the LRU-SP algorithms relieves the problem. On the four disk array, TIP2 incurs an 18% increase in overall execution time due to this additional stall; on eight disks, TIP2's increase is 53%.

### 5.2.3 Small Sequential Reads: AGREP

AGREP reads many small files sequentially with no re-use. As the graph shows, TIP2 performs substantially better on small arrays, and worse on larger arrays. This is due to two competing effects, both related to sorting of elements in the disk queues. As described in Section 4.8, some form of sorting and batching algorithm is critical for good performance. AGREP exhibits some of the tradeoffs. I begin with a quick description of the relevant issues, then describe the actual effects.

An algorithm may submit as many simultaneous prefetches as it wishes up to the size of the buffer cache. The advantage of beginning more than one prefetch at a time is that disk sorting in the driver queue allows the requests to be fetched in an order that is more appropriate for the layout of data on the disk, and therefore has a lower average I/O time. However there are several disadvantages. First, the buffers must be

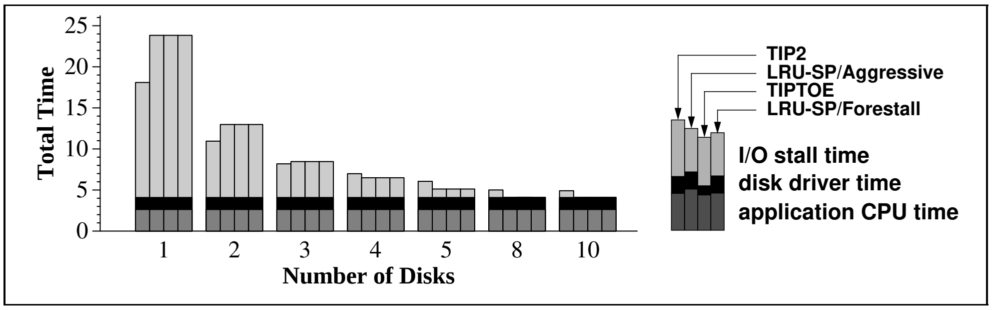



Figure 5.3: Standalone AGREP, four prefetching and cache management algorithms.

allocated before the I/O begins, so each additional prefetch requires an extra buffer that cannot perform useful caching. For AGREP, this is not an important concern because the application has no re-use. Second, if 100 prefetches are begun at once, it is possible that the block that is needed next may be sorted to the very end of the queue, and the system will stall for a hundred disk accesses before the block is brought in. In general, a system must avoid extra stall incurred due to re-ordering of I/O's for needed blocks far into the future. However, this second issue is not relevant for highly I/O bound situations. Every block must be fetched, and there is relatively little computation, so overlapping I/O with computation is less important than keeping the average I/O time down. TIP2's policy naturally addresses this issue in a well load-balanced setting. On smaller arrays for I/O bound applications, TIP2 may keep $\hat{P}$ elements at the queues, resulting in a lower I/O time. But as the array size grows, the $\hat{P}$ elements that are outstanding at the disks will be spread over a larger number of disks and each block may be re-ordered behind only a small number of other blocks. Another advantage of the TIP2 policy, and in fact the reason motivating the decision, is that the current approach is independent of data layout and therefore the allocator need not be aware of details of the I/O subsystem design.

There are a number of other issues to do with management of buffers and eviction decisions once the number of outstanding buffers has been chosen; Section 5.3 discusses these issues in more depth.

In AGREP, TIP2 performs well on three and fewer disks, particularly on one disk. The average queue depth[2] is 34 on one disk, 16 on two disks, and 10 on three disks. On one disk, for instance, the average depth of 34 makes sense because $\hat{P} \cong 68$ and on average, half the blocks in the next $\hat{P}$ elements will already have been read. On two disks, each disk has on average 34 elements within the prefetch horizon, and roughly half of these have been fetched, yielding about 16. Thus, the queue depth for each disk shrinks as the prefetch horizon extends. The average depth drop to 5.8 blocks on five disks, 4.2

[2]The average queue depth is sampled as each request is submitted to a disk, so does not include time when the disk is idle.

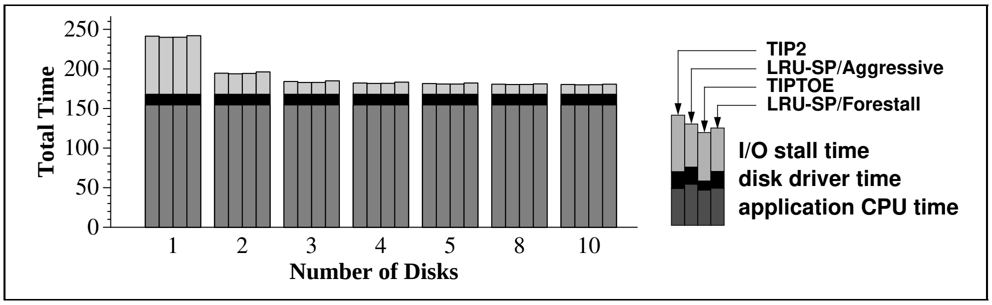



Figure 5.4: Standalone SPHINX, four prefetching and cache management algorithms.

blocks on eight disks, and 3.8 blocks on ten disks (for sequential reads on an extremely large array, this value does not drop to zero because the stripe unit is eight blocks).

The average queue depth for all three of the other algorithms never drops below 7.2 blocks or grows above 8.8 blocks for any array size. Thus, TIP2 achieves a much better average I/O time on smaller arrays (32% lower for a single disk), but a worse average I/O time for larger arrays (6% higher on five disks).

The same effect is visible to a lesser extent when GNULD runs on a single disk.

### 5.2.4 Late-Arriving Hints: SPHINX

As described in Section 3.2, SPHINX is a speech recognition system that performs a viterbi beam search over a large number of language elements. During each 10 ms round of viterbi search, SPHINX gives hints for the nodes that will be examined during the next round. Table 3.5 shows that 72% of the hint batches have ten or fewer hints, and 39% of the batches have only one or two hints. Thus, SPHINX has far fewer pending hints than any other application in the suite.

In the single-process domain, all the algorithms perform well because there is no competing consumer interested in evicting hinted blocks that have been consumed and have no future hints — for all array sizes, the fastest and slowest algorithms differ by no more than 1.3% in overall execution time. In the multi-process case, though, it becomes critical to cache posthint blocks in case future hints arrive for them.

### 5.2.5 Unhinted Accesses: POSTGRES1 and POSTGRES2

In POSTGRES1, TIP2 performs similarly to the other algorithms, and LRU-SP/AGGRESSIVE performs a tiny fraction better. In POSTGRES2, TIP2 performs substantially

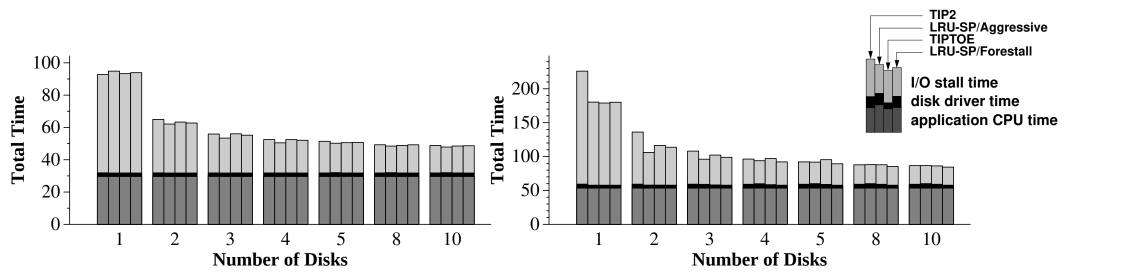



Figure 5.5: Standalone POSTGRES1 and POSTGRES2, four prefetching and cache management algorithms.

worse, and again LRU-SP/AGGRESSIVE performs well. To understand these results, I must review the hinted join operation. The outer relation is unindexed; the inner is indexed. The inner index, and the entire outer relation, can fit together in the cache. POSTGRES1 makes two passes through the outer relation. It begins by hinting a sequential read of the outer relation, then reading the tuples in order and looking them up via unhinted reads to the inner relation index. It saves the addresses of hits in the inner relation. Once the loop completes, it hints the inner relation data reads, then once again loops through the outer relation. It skips the inner relation index reads during this second loop as the addresses of hits have been pre-computed, and goes directly to the inner relation data blocks. For each hit, it writes an output tuple.

There are several opportunities for re-use, which I phrase in terms of TIPTOE's caches. During the first loop, the inner relation index blocks are being re-used from the LRU cache. During the second loop, the outer relation data blocks are being re-used from the posthint cache. And finally, again during the second loop, inner relation data blocks are being re-used from the hinted cache. Thus, all three of TIPTOE's caches have hits. Surprisingly, caching the outer relation data blocks is *not* the correct approach. A back-of-the-envelope calculation confirms this. There are 409 outer relation data blocks, each of which is read once during the second loop. In POSTGRES2 there are about 15,000 inner relation data block reads in the second loop. So using buffers for posthint caching will save 409 reads out of 15,000, allowing each buffer to generate one cache hit. On the other hand, there are 4096 blocks in the inner relation, and 15,000 reads. Therefore, under optimal replacement, each buffer will generate more than 1 cache hit.

POSTGRES1 performs about 4500 reads during the second loop, and touches 2688 inner relation data blocks. Thus, once again, buffers have the opportunity to generate more than one cache hit.

What actually happens in TIPTOE is the following. After the first loop, there are 409

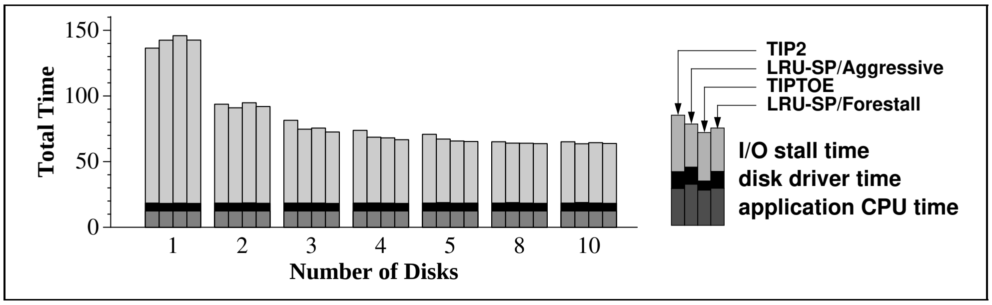



Figure 5.6: Standalone GNULD, four prefetching and cache management algorithms.

buffers in the posthint cache, and the remaining cache buffers hold inner relation index blocks for the LRU. As the second loop begins, the LRU estimator values buffers more highly until the aging catches up, so the posthint, which currently has no hits, begins to shrink. Additionally, as consumption of hinted inner relation data begins, more buffers are placed into the posthint cache, polluting the hit counts and lowering the value.

If the posthint were to remain larger at 409 blocks with no pollution, TIPTOE's local estimate would consider each segment alone, and would therefore estimate that the 50 buffers in the segment holding blocks 400–450 are individually generating the 409 hits. This would assign artificially high value to the posthint estimator, and possibly skew the results. However, the size of the posthint cache has shrunk substantially, so the cost of taking even more buffers from the posthint becomes smaller than the cost of using buffers from the hinted cache.

It remains to discuss TIP2's approach. Hinted cache re-use in Postgres is often for very distant data — the most distant block in the hinted cache will often not be read for more than 10,000 accesses. TIP2's hinted cache estimator (Equation 4.1) assigns low value to these blocks; to avoid this exact problem, TIPTOE's estimator (Equation 4.4) assigns higher value to hinted cache blocks on constrained disks beyond the prefetch horizon.

TIP2 on a single disk caches 4% fewer hinted cache blocks than the other algorithms for POSTGRES1, and 16% fewer for POSTGRES2. On POSTGRES2, these lost hinted cache opportunities cause TIP2 to go to disk 39% more often than TIPTOE, and increase the overall execution by 26% relative to TIPTOE.

### 5.2.6 Multi-Pass: GNULD

GNULD makes several passes through the executable, the last two of which represent most (86%) of the read activity. The first of these two final passes touches 6,370 blocks

```
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
----------------------------------------------------------------------------------------------------
------------------------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxx-xxx----------------------------------------------------------------------
--------------------------------------------------xxxxxx--------------------------------------------
-----------------------------------------------------------------------------------------
```

Figure 5.7: Snapshot of DAVIDSON cache state under TIP2. Each DAVIDSON loop touches 2089 blocks; the figure shows the next 2089 hints (each line contains 100 hints) rendered as a "-" if the hinted block is cached and an "x" if the block is missing. TIP2 clusters all its missing blocks together into a single, long, sequential run.

of which about 20% have been seen before. The second pass touches 4,356 blocks, of which 79% are re-used. However, the hints for this high-reuse pass are given immediately before the pass itself, so any blocks still in memory must be in an unhinted cache.

The differences between algorithms are most apparent on a single disk. First, like AGREP, GNULD benefits from deep queues. TIP2 performs more sorting and generates a lower average I/O time on a single disk: 9% lower than the other algorithms.

As in AGREP, TIP2's caching benefit remains significant on two disks, then becomes a disadvantage on larger arrays, for which $\hat{P}$ total outstanding prefetches corresponds to fewer elements in each disk queue than the other algorithms.

## 5.3 Single-Process Issues: Batching and Re-Use

This section presents a more concrete analysis of the differences between batching policies. To recap, TIP2 will typically submit a request for any missing block within the next $\hat{P}$ hints. The other algorithms submit up to 16 requests to a disk whenever it goes idle, and never submit to a disk that is not idle. DAVIDSON's looping structure makes it the natural trace for examining this mechanism.

```
-----------------------------------------------------------------------------------xxxxxxxxxxxx-----
----------------------------------------------------------------------------xxxxxxxxxxxxxxxxx-------
------------------------------------------------------------------xxxxxxxxxxxxxxxxx-----------------
-----------------------------------------------xxxxxxxxxxxxxxxxx------------------------------------
----------------------------------xxxxxxxxxxxxxxxxx-------------------------------------------------
--------xxxxxxxxxxxxxxxxx---------------------------------------------------------------xxxxxxxxxxxx
xxxxx-------------------------------------------------------------------------------xxxxxxxxxxxxxxxx
x-----------------------------------------------------xxxxxxxxxxxxxxxxx-----------------------------
-------------------xxxxxxxxxxxxxxxxx------------------------------------------------------xxxxxxxxxx
xxxxxxx--------------------------------------------------xxxxxxxxxxxxxxxxx--------------------------
-------------------------------xxxxxxxxxxxxxxxxx-------------------------------xxxxxxxxxxxxxxxxx----
---------------------------------------------xxxxxxxxxxxxxxxxx--------------------------------------
--------------xxxxxxxxxxxxxxxxx--------------------------------------xxxxxxxxxxxxxxxxx--------------
------------------------------xxxxxxxxxxxxxxxxx----------------------------------------------xxxxxxx
xxxxxxxxxx-----------------------------------------xxxxxxxxxxxxxxxxx--------------------------------
------------------------xxxxxxxxxxxxxxxxx--------------------------------------------xxxxxxxxxxxxxxx
xx---------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx-xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx-xxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxx---------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx-------------------------
```

Figure 5.8: Snapshot of DAVIDSON cache state under TIPTOE. Each DAVIDSON loop touches 2089 blocks; the figure shows the next 2089 hints (each line contains 100 hints) rendered as a "-" if the hinted block is cached and an "x" if the block is missing. TIPTOE's batching policy distributes missing blocks in clusters throughout the sequence.

To show the state of the cache I show snapshots taken during the execution of the DAVIDSON trace in different scenarios. Every snapshot is taken immediately before processing line 20,000 of the trace file. The snapshots show the next 2089 hints in order — missing elements are represented by "x"s and cached elements by "-"s. Figure 5.7 shows a snapshot of TIP2's execution. The missing blocks are clustered together into a single region. During each pass through the data, prefetching for these missing blocks begins $\hat{P}$ elements before the beginning of the block, evicting the most recently consumed block for each prefetch. Thus, the missing section of the trace moves back by $\hat{P}$ elements during each iteration.

In general, this type of clustering has the advantage that locality in the trace tends to correspond to disk locality. In this case, it doesn't yield better sorting as the trace is already sequential, but it does allow the disk to retrieve sequential blocks without having to perform seeks. On the other hand, this layout also results in TIP2 allowing the disk to go idle for the longest possible fraction of the sequence — if the missing blocks were distributed through the sequence then some missing block would be within the next $\hat{P}$ requests a greater fraction of the time.

Figure 5.8 shows the same situation under TIPTOE. Batching in this situation always submits a full batch of requests, evicting the most recently consumed 16 blocks. These 16 requests complete at the disk, and then the next batch is constructed. This results in

```
----------------------------------------------------x--------------x--------------------x-----------
x--------------------------------x------x-------------------------x------------x-------------------x
--------------------x---------------------------x--------------------------x------------------------
----------x-------------------------x-------------------x-------------------x-----------------------
---------x--------------------------x--------------------x------------------x-----------------------
---x------------x-------------x-------------------x-------------------x-------------------x---------
-----------x-------------------x-------------x-------------------x----------------------------x-----
------x---------------------x------------x-------------x--------------------x-----------------------
-----x-----x-----------------x---------------------x--------------------x------------x--------------
------x-------------x------------x--------------------------x-------------x-------------------x-----
--------------x-------------------x-------------------x-------------------x------------x------------
--------x------------x------------x-------------x-------------------x-------------x------------x----
----------------x-------------------x-------------x-------------x--------------------x--------------
x-----------x---------x-----------xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxx-xxxxxxxxxxxxxxxxxxxx-xxxxxxxxxxxxxxxxxxxx-xxxxx-xx--x------
```

Figure 5.9: Snapshot of DAVIDSON cache state under TIPTOE without batching. Each DAVIDSON loop touches 2089 blocks; the figure shows the next 2089 hints (each line contains 100 hints) rendered as a "-" if the hinted block is cached and an "x" if the block is missing. With a policy of keeping 16 elements at each disk queue rather than allowing the queues to drain until empty, a deep prefetching algorithm distributes missing blocks uniformly throughout the sequence.

the well-distributed clusters of missing blocks in the figure. If the disk were able to keep up with the demands of the application, the clusters of missing blocks would cover the entire sequence; however, the application is on average I/O bound. Thus, at some point, there remains a stream of missing data and the application stalls.

The advantage of such a layout is that each cluster of 16 missing blocks has sufficient locality for quick fetching — I refer to this phenomenon as "re-fetch" locality because the algorithm evicted data that must be fetched again later in such a way as to make the later fetch efficient. Batching in this situation has good re-fetch locality due to the sequentiality of the sequence. In general, however, it might be preferable to alter eviction decisions based on the hint sequence so as to cluster evictions for re-fetching, even if the missing blocks do not occur sequentially in the sequence. I do not address this more general problem.

Another interesting property emerges from this layout. When TIPTOE begins reading through the section of the trace in which missing clusters are distributed nicely, the I/O and compute demands of that portion of the sequence are exactly matched. This occurs because the next batch is submitted only when the previous batch has completed, so enough cached data is consumed to allow the batch to complete (and no more).

Another natural policy might be to avoid batching because it does not keep the disk queue full — it fills the queue and then allows it to drain. Instead, consider the policy of keeping 16 elements sorted in the queue at all times, and submitting a new one whenever one completes. One obvious disadvantage of such a scheme is fairness: any particular request may be delayed for a long time before it completes. However there is a more subtle effect that may be more important. Figure 5.9 shows TIPTOE modified to keep batches full at all times. The resulting layout essentially distributes clusters of missing blocks of size 1 throughout the sequence. Again, the property exists that the nicely-distributed portion of the trace has matched I/O and computation. But the I/O now takes substantially longer because the locality of those evenly-distributed single missing blocks is poor compared to the clusters of missing blocks in the other schemes.

## 5.4 The Single-Process Case: Lessons Learned

This section contains a distillation of the results above into a number of lessons learned in the course of the work. There are four lessons that apply to the SPACE problem alone. That is, these lessons apply even if there are no unhinted accesses, and all hints are available when execution begins.

**Lesson 1:** *Leaving a constrained disk idle leads to additional stall.* This effect is described in detail in Figure 4.3 and the accompanying text. It, and Lesson 2, motivated the development of FORESTALL. The graph for DAVIDSON is the most telling example of this problem.

**Lesson 2:** *Submitting an I/O requires $T_{\text{driver}}$ computational overhead.* This effect is described in detail in Figure 4.4, and the accompanying text. The problem arises whenever high bandwidth and high re-use exist. Again, the effect is most visible in the DAVIDSON trace with larger array sizes.

The next two lessons are specific to queueing.

**Lesson 3:** *Deeper disk queues yield lower average disk service times.* All four systems are based on a system model that assumes a constant disk service time, so modeling of queue sorting and locality is beyond the scope of theoretical analysis. However the simulator performs CSCAN sorting in the queues, and the disk simulator includes such non-constant effects as disk geometry and on-disk readahead buffering. Therefore the policy used by each prefetching algorithm to determine when exactly to submit prefetches to the disk driver has significant effects on the overall execution time. TIP2's policy is to submit prefetches out to the prefetch horizon; thus, TIP2 will commonly keep $\hat{P}$ buffers at the disk queue. The other algorithms submit up to sixteen requests whenever the disk goes idle in order to attain the benefits discussed in Lesson 4, but in doing so they typically generate shorter disk queues for small arrays. This effect is most visible in the AGREP application, and to some extent in GNULD. As discussed under those applications, as the array size grows, TIP2's policy eventually induces shorter queues than the other algorithms', with the split occurring around three or four disks.

**Lesson 4:** *Eviction decisions impact locality of "re-fetched" data.*

When a prefetching algorithm fetches data and evicts blocks that will be needed again, it may attempt to select blocks for eviction so as to increase disk locality when the blocks are read back in. This topic is discussed in more detail in Section 5.3. Briefly, Cao *et al.* [CFKL95b] describe a mechanism they call "batching" in which the prefetching algorithm waits for the disk to go idle and then submits up to $B$ requests, where the batchsize $B$ is a parameter of the algorithm (16 in my implementation and Cao's). LRU-SP/AGGRESSIVE, TIPTOE and LRU-SP/FORESTALL all adopt this scheme. On cyclic datasets, the $B$ evicted elements are typically the most recently consumed blocks. Since neighboring blocks in the access stream display locality on the disk, this scheme allows the blocks to be re-fetched with low average disk service time. The batching heuristic provides good re-fetch locality whenever there is re-use in a sequential access pattern.

# Chapter 6

# Multi-Process Informed Prefetching and Caching

*It is a very hard undertaking to seek to please everybody.*

— *Publius Syrus, "Maxim 675"*

This chapter extends the evaluation of Chapter 5 into the domain of multiple processes executing simultaneously, each disclosing an arbitrary fraction of its accesses in the course of execution. The work described here is an extension of work presented at Sigmetrics 97 in collaboration with Hugo Patterson and Garth Gibson [TPG97].

I begin with a discussion of metrics for the multi-process case in Section 6.1. Next, in Section 6.2 I study pairs of the applications described in Chapter 3 running simultaneously. Then Section 6.3 considers a similar situation in which resource contention arises: an I/O-intensive process running in the presence of background load. Section 6.4 presents a more detailed analysis of particular issues that often arise in the multi-process case. Finally, Section 6.5 presents a distillation of the lessons learned in the course of performing these multi-process experiments.

## 6.1 Multi-Process Metrics

In the single-process case, total elapsed time is a metric with a clear, unambiguous meaning. In the multi-process case, while it is still a useful metric, other issues intrude. Imagine running traces of application $A$ and application $B$ side-by-side. One algorithm (call it the *unbalanced algorithm*) might choose to dedicate all its resources to application $A$ until $A$ completes, then hand all the resources to application $B$. Another algorithm (call it the *balanced algorithm*) might split resources evenly between both applications until they both complete.

If the applications are I/O bound and give reasonable hints, so that the disks can be kept busy, then total time will be I/O time, and will correspond to number of cache hits. The unbalanced algorithm will have more cache hits since it dedicates more buffers to whichever application it happens to be running.

If the applications are compute bound and give good hints then the processor will not have to wait for I/O, so complete time will be equal to compute time. In this case, the unbalanced algorithm will again perform better since there will be fewer I/O's, and thus less driver overhead.

On the other hand, if the total execution time of the unbalanced algorithm is greater than the max of compute time and I/O time (*i.e.*, if there is both computation time that is not overlapped with I/O, and I/O time that is not overlapped with compute) then the balanced algorithm will have the opportunity to overlap one application's compute with the other application's I/O, and vice versa. The tradeoff is therefore how much additional overlap can be created, versus how much additional I/O is required due to the split of resources between the two applications.

In summary, if one algorithm gives a better overall execution time for both applications than another algorithm does, it is important to discover the true source of the difference. It is possible that the faster algorithm made better use of the resources for each application, passed buffers back and forth at appropriate times, and so on. On the other hand, it is also possible that the faster algorithm simply induced an unfair split of resources that happened to result in a shorter time-span. Such a decision would not necessarily be correct due to the reduction in fairness.

Table 6.1 gives the relative execution times of all pairs of processes. These values are computed as follows. The single-process time for each application is fixed to be the time required by that application under TIP2 on a single disk (arbitrarily). The value in the table under DAVIDSON/XDS, for instance, is the execution time of the faster as a percentage of the execution time of the slower. In this case the value is 82%, since XDS takes 82% of the time DAVIDSON takes. Thus, if this pair were to run alongside one another, both could run for a substantial fraction of the overall time. The rows and columns are ordered by decreasing total time, so the application named in the column is longer than the application named in the row.

## 6.2 Two-Process Experiments

To create a set of experiments that are as fair as possible I chose pairs of applications that are most similar in overall execution time. I rejected any pair of applications in which the faster completed more than twice as fast as the slower. This resulted in 11 experiments, which I now study in detail.

| | Davidson | Xds | Sphinx | Postgres2 | Gnuld | Postgres1 | Agrep |
|---|---|---|---|---|---|---|---|
| Davidson | 100% | | | | | | |
| Xds | 82% | 100% | | | | | |
| Sphinx | 62% | 76% | 100% | | | | |
| Postgres2 | 50% | 61% | 81% | 100% | | | |
| Gnuld | 34% | 41% | 54% | 67% | 100% | | |
| Postgres1 | 25% | 31% | 41% | 50% | 75% | 100% | |
| Agrep | 13% | 16% | 21% | 26% | 39% | 51% | 100% |

Table 6.1: Relative Execution Times for Pairs of Traces

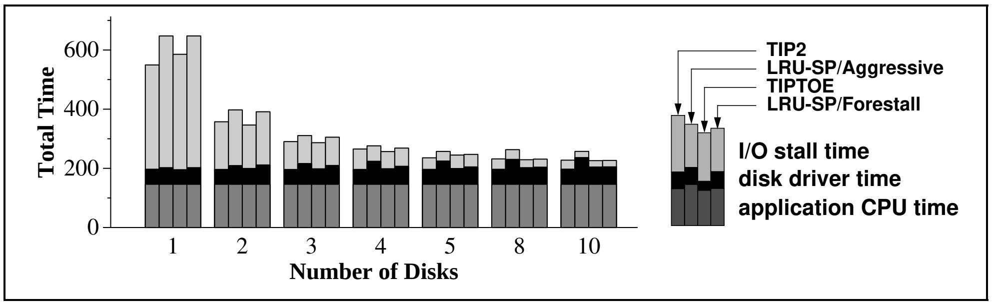



Figure 6.1: Experiment 1: Davidson/Xds, four prefetching and cache management algorithms.

### 6.2.1 Davidson/Xds

The Davidson/Xds graph shows some computational overhead in the 8 and 10 disk cases for lru-sp/aggressive; other than that, the primary differences are visible on one and two disks, and slightly visible on three and four disks. The lru-sp-based algorithms cache 69000 blocks; the cost-benefit algorithms cache 79500 (TIP2) and 82800 (tiptoe). Both Davidson and Xds consume data rapidly, but Xds exhibits little re-use while Davidson exhibits strong re-use. As the disk is always constrained, both lru-sp/aggressive and lru-sp/forestall prefetch aggressively. And since Xds consumes data quickly but has less re-use than Davidson, lru-sp will sometimes evict useful Davidson blocks in favor of less useful Xds blocks.

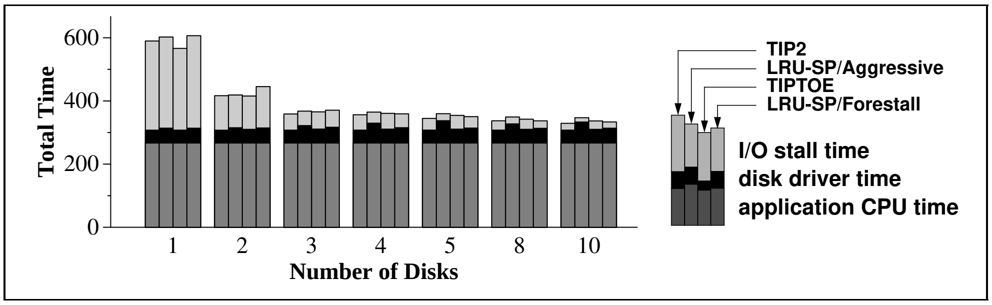



Figure 6.2: Experiment 2: DAVIDSON/SPHINX, four prefetching and cache management algorithms.

### 6.2.2 DAVIDSON/SPHINX

In the DAVIDSON/SPHINX graph three primary effects are visible. First, LRU-SP/AGGRESSIVE flushes the DAVIDSON cache on larger array sizes as described in Section 5.2.1. Second, on 2 disks LRU-SP/FORESTALL completes later than the other algorithms. The COST-BENEFIT algorithms cache more effectively for DAVIDSON than the LRU-SP algorithms; TIPTOE caches 7% more blocks than LRU-SP/FORESTALL, for example. LRU-SP/AGGRESSIVE outperforms LRU-SP/FORESTALL due to deeper prefetching for SPHINX, because the number of pending hints given by the SPHINX trace is often small (we examine this fact in more detail under the SPHINX/GNULD experiment), and the compute time is quite large, so LRU-SP/FORESTALL believes the disks are not constrained until a later batch of hints arrives. Finally, on a single disk the COST-BENEFIT algorithms perform well by caching more effectively for DAVIDSON, and TIPTOE outperforms TIP2 by 4% due to deep prefetching for DAVIDSON.

### 6.2.3 XDS/SPHINX

The XDS/SPHINX graph shows LRU-SP/AGGRESSIVE and TIPTOE performing 9% worse than the other algorithms on a single disk. TIP2 and LRU-SP/FORESTALL both prefetch less aggressively for SPHINX than for XDS, and therefore complete the XDS trace more quickly. This results in a 15% better I/O time for these two algorithms, which results in the difference visible in the graphs. The detailed reason that TIP2 and LRU-SP/FORESTALL prefetch less aggressively for SPHINX, and that the I/O time increases as a result, is the following. First consider TIPTOE and LRU-SP/FORESTALL. 380 seconds into the trace, TIPTOE has completed 82000 lines of XDS, and 123000 lines of SPHINX. LRU-SP/FORESTALL has also completed 82000 lines of XDS, but only 112000 lines of SPHINX by dedicating cache buffers unnecessarily to XDS. Before TIPTOE completes XDS, SPHINX

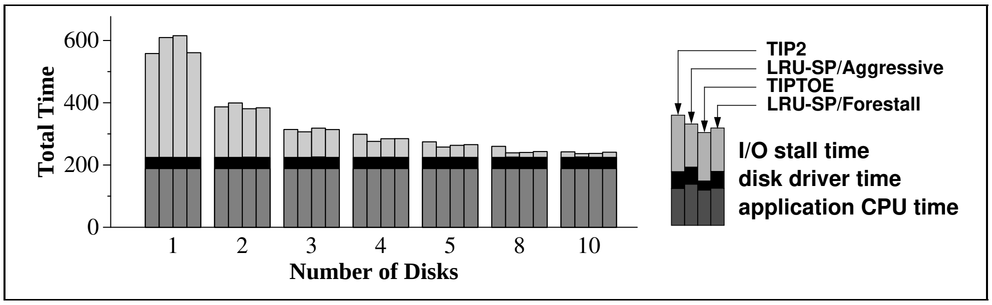



Figure 6.3: Experiment 3: XDS/SPHINX, four prefetching and cache management algorithms.

leaves its initial phase of hinted sequential reading, and begins 2000 unhinted reads. This sequence arrives while XDS continues to execute, resulting in an effect I consider in more detail in Section 6.3.1. Briefly, demand misses take priority over prefetches, even prefetches that have been promoted to be demand reads, under the assumption that prefetches should not be re-ordered at promotion time because this may destroy locality in the prefetch stream. Thus, when XDS performs prefetching in parallel with SPHINX's unhinted accesses, each SPHINX access will draw the disk head away from XDS' dataset, then as the SPHINX data is being consumed the head will return to prefetching for XDS. The overall average I/O time jumps from 6.4 ms to 7.4 ms over the course of these 2000 accesses, resulting in the disparity visible in the graph. TIPTOE and LRU-SP/AGGRESSIVE essentially perform too well, completing too much of the SPHINX trace before XDS completes, and entering a phase of SPHINX in which it is difficult to overlap two different processes.

The only other effects visible in this trace are familiar: LRU-SP/AGGRESSIVE performs well on SPHINX due to deep prefetching on disks that do not appear to be constrained (see Section 5.2.4), and TIP2 performs poorly on XDS for power-of-two array sizes (see Section 5.2.2).

### 6.2.4 XDS/POSTGRES2

The graph of XDS/POSTGRES2 shows a significant advantage to the COST-BENEFIT algorithms. It also shows that LRU-SP/AGGRESSIVE and LRU-SP/FORESTALL perform similarly, with LRU-SP/FORESTALL performing slightly better; likewise, TIP2 and TIPTOE perform similarly with a slight advantage to TIPTOE. Here the allocation algorithm has a more significant impact than the per-process prefetching algorithm. XDS has minimal re-use and few unhinted requests. POSTGRES2, on the other hand, requires only 9,000 reads and 1,100 demand misses with proper caching. The COST-BENEFIT algo-

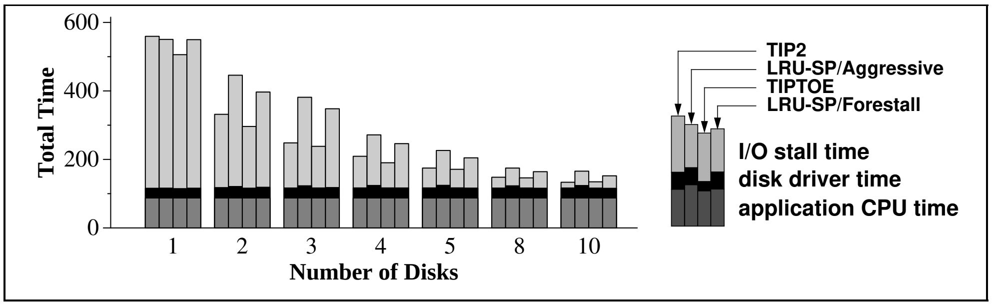



Figure 6.4: Experiment 4: XDS/POSTGRES2, four prefetching and cache management algorithms.

rithms dedicate the cache to unhinted requests in the early section of the POSTGRES2 trace (774/1280 buffers for TIPTOE), and then return most of the cache to buffering hinted blocks once POSTGRES2 completes its phase of unhinted reads to the inner relation, and the estimators note that the LRU cache is no longer providing significant benefit. LRU-SP/AGGRESSIVE and LRU-SP/FORESTALL also display some thrashing, in which one process loads a block that another process evicts. This is not the major factor determining the results, but does represent extra load on the disks, contributing to the stall for intermediate arrays sizes (we examine this effect in more detail under the graph of POSTGRES2/GNULD). With larger arrays (8 and 10 disks) this eviction of unread data is not a substantial factor, but the LRU-SP-based algorithms devote fewer buffers to caching unhinted data, and therefore suffer more demand misses from the POSTGRES2 trace (LRU-SP/FORESTALL for instance incurs 83% more demand misses than TIPTOE on 10 disks), whose latency cannot be masked by prefetching.

### 6.2.5 SPHINX/POSTGRES2

In the SPHINX/POSTGRES2 graph the COST-BENEFIT algorithms again determine correctly that buffers should be dedicated to caching unhinted data during the early portion of the POSTGRES2 trace, and then should be returned to the hinted cache for the remainder of execution. TIP2 outperforms TIPTOE on a single disk by keeping the disk queue more full rather than submitting a batch and then allowing it to drain (9% improvement in average I/O time). If TIPTOE is modified to submit a new batch as necessary to keep the queue at the same size used by TIP2, the performance of the two algorithms is within 1%.

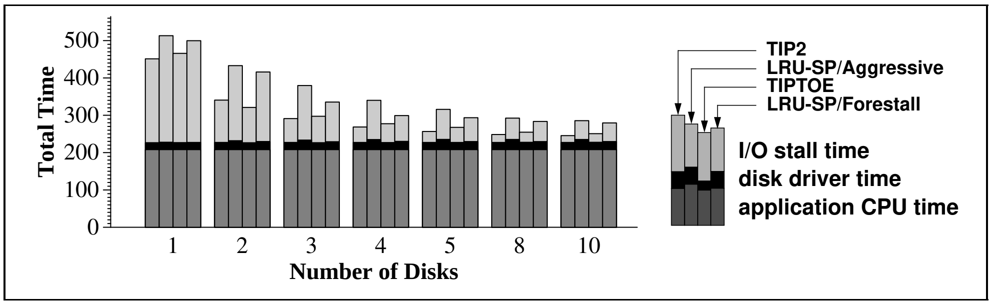



Figure 6.5: Experiment 5: SPHINX/POSTGRES2, four prefetching and cache management algorithms.

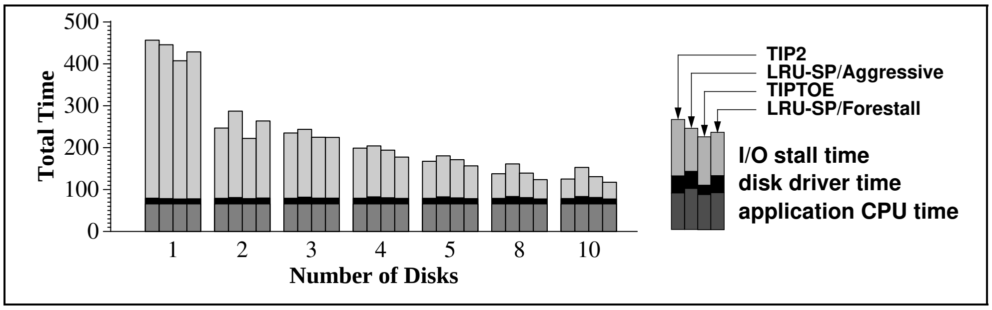



Figure 6.6: Experiment 6: POSTGRES2/GNULD, four prefetching and cache management algorithms.

### 6.2.6 POSTGRES2/GNULD

Once again, in the POSTGRES2/GNULD graph the COST-BENEFIT algorithms perform well in general. TIP2 and TIPTOE perform almost identically for four or more disks, but TIPTOE performs better on smaller arrays (12% better on 1 disk, 10% better on 2 disks, and 5% better on 3 disks). TIPTOE's improved hinted cache estimator results in 15% more cache hits for the single-disk case versus TIP2, with no substantial decline in LRU performance.

LRU-SP/FORESTALL takes 25% more cache misses than TIPTOE on a single disk, and substantially more for intermediate array sizes (2.74 times as many on 2 disks, 61% more on 3 disks, 58% on 4 disks). TIPTOE then incurs 18% *more* misses on ten disks. The reason for these disparities is that all algorithms except TIP2 suffer in this experiment from a thrashing problem, prefetching data, and then evicting it before it can be read. All the algorithms have safeguards against evicting prefetched data from one process to prefetch for another process, but when a demand read arrives, a buffer must be generated.

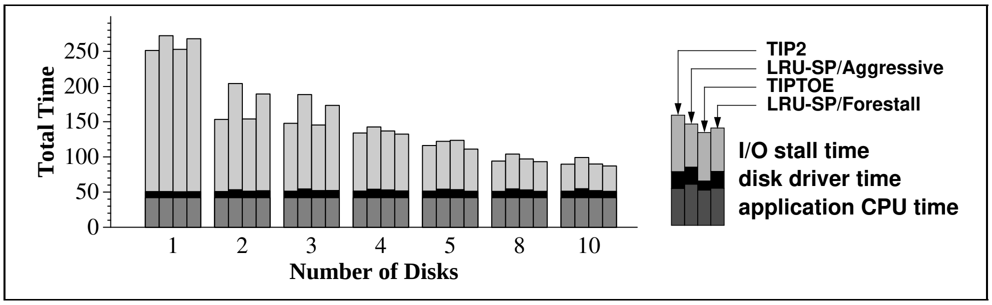



Figure 6.7: Experiment 7: POSTGRES1/GNULD, four prefetching and cache management algorithms.

For LRU-SP algorithms, since POSTGRES2 performs a large number of demand reads early in the sequence, and GNULD processes its data more quickly, the LRU-SP allocator often asks GNULD to provide a block for POSTGRES2. Since GNULD has just been prefetching data, the block it chooses to give up is the block it has most recently read, which results in evicting blocks that have not yet been used by any application, and therefore in additional driver overhead to reload the block later.

Other traces exhibit some thrashing, but not as markedly as here, as both POSTGRES2 and GNULD exhibit substantial non-sequential hinted re-use. It is possible for both processes to prefetch a distant block, consume cached data, and still have the prefetched block be the best eviction decision.

### 6.2.7 POSTGRES1/GNULD

There are three effects visible in this graph. The most significant effect is that COST-BENEFIT allocation performs better than LRU-SP allocation. The second effect is that LRU-SP/AGGRESSIVE performs more fetches on larger arrays and incurs some additional driver overhead. Finally, TIP2 and LRU-SP/FORESTALL beat TIPTOE on 4 and 5 disks.

The POSTGRES1 trace shares the overall structure of the POSTGRES2 trace: the first section of the trace contains unhinted accesses to the inner relation's index. The second section contains hints for accesses to the unindexed relation, and the final section contains hinted accesses to the outer relation. However, the POSTGRES1 trace is significantly smaller since the fraction of tuples that hit in the outer relation is 20% rather than 80%. In the POSTGRES1/GNULD graph the COST-BENEFIT algorithms again dedicate the cache to holding unhinted reads from POSTGRES1 during the first section of the trace.

On larger arrays, about 25% of LRU-SP/AGGRESSIVE's extra fetches are due to will-

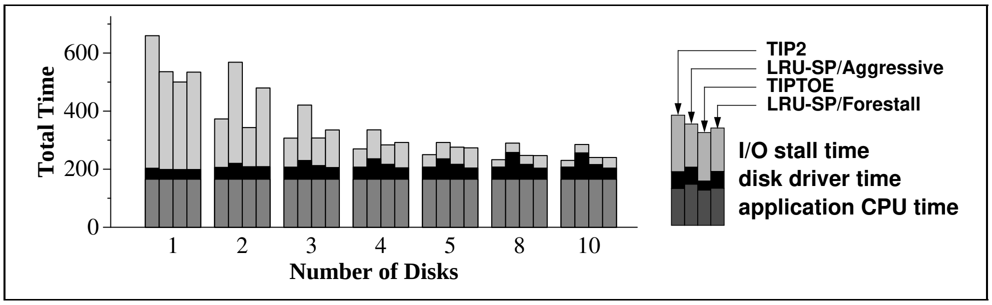



Figure 6.8: Experiment 8: DAVIDSON/POSTGRES2, four prefetching and cache management algorithms.

ingness to evict data from the hinted cache in order to prefetch earlier missing blocks. The remaining 75% are due to eviction of unread blocks from one process in order to prefetch for the other process; these reads make up 21% of LRU-SP/AGGRESSIVE's I/O and increase overall execution time by 5–12%. Finally, on 4 and 5 disks, algorithms other than TIP2 are again displaying some thrashing for the same reasons described under POSTGRES2/GNULD.

### 6.2.8 DAVIDSON/POSTGRES2

There are many effects visible in this trace. First, we consider the single disk case. DAVIDSON is running alongside POSTGRES2. The LRU-SP-based algorithms incur more demand misses (56–64% more than TIPTOE) because they do not cache for POSTGRES2's substantial re-use of the unhinted inner relation index blocks. This trace magnifies the effect because DAVIDSON's consumption rate is so much larger than POSTGRES2's, and LRU-SP-based algorithms partition the cache based roughly on consumption rate. TIP2's performance on a single disk is worse than the other algorithms for two reasons. First, its hinted cache estimator assigns low value to blocks that are cached for re-use outside the prefetch horizon. That is, even on a constrained disk, TIP2's cost of evicting a block that is 100 accesses away is $T_{\text{driver}}/100$, as opposed to TIPTOE's much larger estimate of $T_{\text{disk}}/100$. So TIP2 performs much less caching for DAVIDSON, resulting in 14% more I/O's overall than LRU-SP/FORESTALL. Second, TIP2 performs no batching, and thus evicts blocks from the DAVIDSON trace without clustering for later reading. This effect does not arise when DAVIDSON runs standalone because TIP2 does not evict data from the hinted cache while consuming the cached portion of the sequence — another process running alongside *does* evict data during this part of each pass. The batching algorithms, on the other hand, evict clusters of blocks which for sequential datasets result in greater disk locality when the blocks are read back in. This results in a 9–16% increase in average

I/O time for TIP2 versus the other algorithms. The LRU-SP-based algorithms perform equivalently to within a fifth of a percent, as the disk is almost always constrained. TIPTOE performs better than both algorithms by dynamically resizing the cache during the different phases of POSTGRES2's execution. It maintains about 700 buffers in the LRU during the first phase, in which the inner relation index blocks see high re-use. During this phase, the outer relation data blocks are being placed into the posthint queue, but there is no re-use until the second phase begins. As DAVIDSON is aggressively reading throughout this phase, these blocks are evicted to cache DAVIDSON data blocks. During the second phase, in which POSTGRES2 reads the outer relation again and the hinted inner relation data blocks, TIPTOE does not hold outer relation blocks in the posthint. This is the correct decision: the relative rates of the processes makes caching for DAVIDSON a better use for these blocks. For the remainder of the execution, the LRU and the posthint caches are empty.

On 2 and 3 disks, both processes are still constrained; COST-BENEFIT outperforms LRU-SP and on 2 disks TIPTOE outperforms TIP2. The latter effect is due to deeper prefetching on TIPTOE's part, which results in 10000 more prefetches completed before the associated read, and 2200 fewer prefetches in progress when the read arrives, compared to TIP2. The LRU-SP-based algorithms do not cache for POSTGRES2 and suffer a substantial increase in demand misses; LRU-SP/FORESTALL for instance incurs more demand misses than TIPTOE by a factor of 7 on 2 disks, and a factor of 4 on 3 disks. Again, this effect arises because LRU-SP chooses to prefetch data for quick-running DAVIDSON to the detriment of POSTGRES2's accesses to the LRU cache.

This pair of traces has the property that running each process independently tends to perform better than multitasking. Running each trace independently is 20% faster than the best combined run on a single disk, 19% faster on 3 disks, and roughly similar on 5 disks. Thus, TIP2's over-valuation of the LRU cache allows it to finish POSTGRES2 earlier, and then dedicate the cache entirely to DAVIDSON, resulting in a better overall execution time.

LRU-SP/AGGRESSIVE also displays a significant amount of thrashing — 21% of all blocks must be read twice for LRU-SP/AGGRESSIVE, and 3% for LRU-SP/FORESTALL, on 10 disks. This thrashing occurs because these algorithms all read data as aggressively as possible for DAVIDSON, at the same time that demand reads are arriving for POSTGRES2, which results in eviction of a substantial fraction of the newly-read blocks.

### 6.2.9 SPHINX/GNULD

The most striking effect of the SPHINX/GNULD graph is that performance is better on three disks than on four or five. This is not due to load balancing; it is a sequentiality effect. SPHINX typically gives small batches of hints. There are 113 batches containing

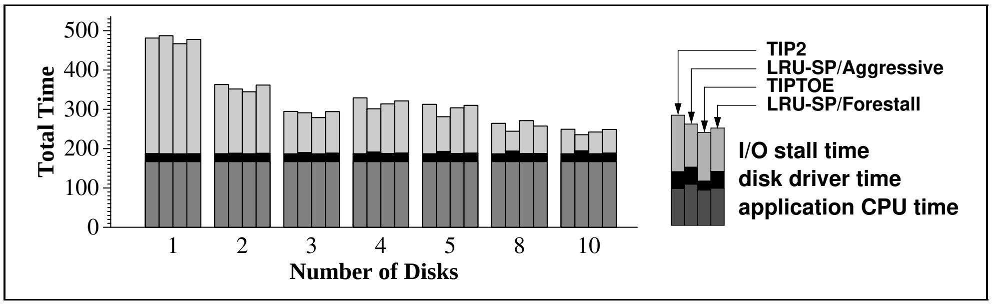



Figure 6.9: Experiment 9: SPHINX/GNULD, four prefetching and cache management algorithms.

449 hints, then 50 other batches containing more than 64 hints, and the remainder of the batches (41% of the total batches) follow the distribution shown in Figure 6.10. The stripe unit of the drives in our simulator is eight blocks. Therefore, on two or three drives we usually have sufficient hints to submit two stripes, or one full batch, to each drive. But when we reach four or five disks, this is not always the case and the average I/O time increases. Despite having plenty of I/O bandwidth, the hints are arriving late enough that latency is important, so for instance in the case of TIPTOE the average I/O time increases by 5%, resulting in a 5% increase in total execution time. The other primary visible effect is that, as described in Section 5.2.4, LRU-SP/AGGRESSIVE generally performs well when the hint sequence becomes very short.

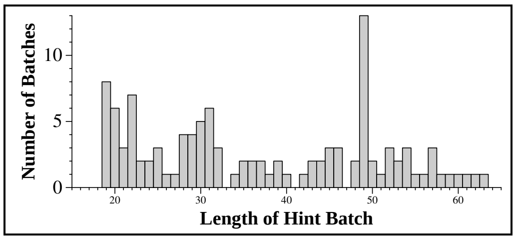



Figure 6.10: Distribution of hint batch sizes for the SPHINX trace.

### 6.2.10 POSTGRES2/POSTGRES1

The POSTGRES2/POSTGRES1 trace shows two processes with substantial numbers of demand reads competing with one another. On a single disk TIPTOE performs substantially

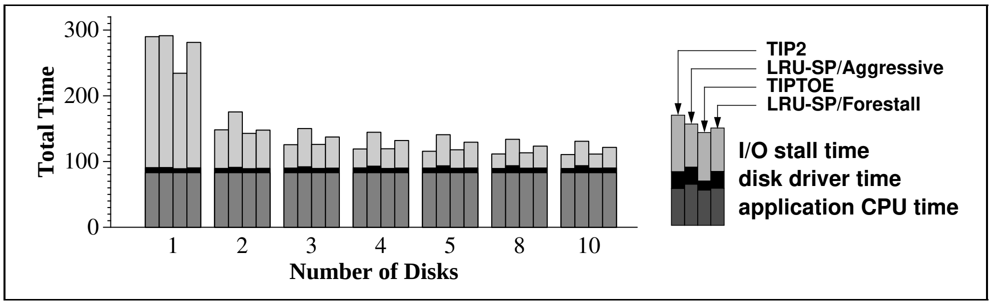



Figure 6.11: Experiment 10: Postgres2/Postgres1, four prefetching and cache management algorithms.

better than TIP2 because once Postgres1 begins its hinted read phase, Postgres2 is still performing unhinted reads. Tiptoe's hinted cache values the hinted blocks of the Postgres1 trace correctly with respect to the unhinted blocks still being read by Postgres2, but TIP2 evicts blocks from Postgres1's hinted cache. This increases the number of demand misses, and increases the amount of stall for prefetched blocks that are still busy when the read arrives. The lru-sp-based algorithms do not allocate sufficient resources to LRU caching once the hinted phases begin (as there are still some unhinted reads during these phases), and lru-sp/forestall suffers an additional 18% demand misses compared to tiptoe on a single disk, 28% on two disks, increasing to 72% on 10 disks. On larger arrays, the lru-sp-based algorithms again incur demand misses whose latency cannot be masked; lru-sp/forestall does not prefetch as aggressively when the disks are constrained, so is not as heavily influenced as lru-sp/aggressive.

### 6.2.11 Postgres1/Agrep

Agrep hints all its reads, but Postgres1 contains an initial segment of unhinted reads, and includes some unhinted reads throughout the trace. On all array sizes the cost-benefit algorithms dedicate more of the cache to holding unhinted data and thus incur fewer demand misses; on one and two disks this difference is a substantial effect. On two or more disks, the additional demand misses explain the additional stall. On one disk, however, lru-sp/aggressive takes 25 seconds longer to complete than tiptoe (21%). 8 seconds of this additional stall are explained by additional demand misses incurred by lru-sp/aggressive. 10 more seconds are explained by hinted cache management decisions. The lru-sp-based algorithms dedicate buffers to Agrep, which has low re-use, and to Postgres1, which has higher re-use, based on their relative rates which are quite similar. The cost-benefit algorithms on the other hand prefer to give buffers to Postgres1, and increase the amount of caching of hinted data. Tiptoe caches 11%

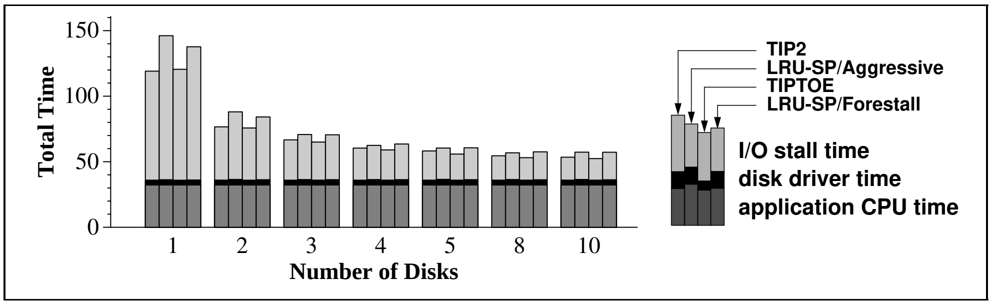



Figure 6.12: Experiment 11: POSTGRES1/AGREP, four prefetching and cache management algorithms.

| Disks | TIP | LRU-SP/AGG | LRU-SP/Forestall |
|---|---|---|---|
| 1 | 1.053 | 1.097 | 1.070 |
| 2 | 1.047 | 1.235 | 1.162 |
| 3 | 1.013 | 1.170 | 1.099 |
| 4 | 1.014 | 1.103 | 1.044 |
| 5 | 0.984 | 1.072 | 1.025 |
| 8 | 0.995 | 1.093 | 1.017 |
| 10 | 0.993 | 1.109 | 1.028 |
| * | 1.014 | 1.124 | 1.063 |

Table 6.2: Summary of results for two hinting processes. This table gives ratio of elapsed time for an algorithm to the elapsed time for TIPTOE. Numbers are the geometric mean of the ratios for the eleven experiments. The last line, marked with a '*', is the mean over all array sizes.

more blocks resulting in the 10 saved seconds of I/O. Finally, the remaining 7 seconds are due to a 10% faster average I/O time for TIPTOE. This in turn is due to the unhinted reads that cause the head to move from processing a run of sequential reads from AGREP in order to fetch a higher priority demand miss for POSTGRES1. If the priorities are eliminated the I/O time is reduced by 35% for LRU-SP/AGGRESSIVE, but this is at the cost of significant increase in stall time for POSTGRES1.

| Disks | 1 | 2 | 3 | 4 | 5 | 8 | 10 | Totals |
|---|---|---|---|---|---|---|---|---|
| Exp1 | 1.106 | 1.130 | 1.066 | 1.047 | 1.012 | 1.009 | 1.000 | 1.052 |
| Exp2 | 1.071 | 1.072 | 1.014 | 0.997 | 0.989 | 0.985 | 0.991 | 1.016 |
| Exp3 | 0.911 | 1.008 | 0.984 | 1.000 | 1.008 | 1.012 | 1.017 | 0.991 |
| Exp4 | 1.087 | 1.338 | 1.458 | 1.295 | 1.193 | 1.123 | 1.134 | 1.226 |
| Exp5 | 1.073 | 1.297 | 1.128 | 1.079 | 1.097 | 1.114 | 1.116 | 1.127 |
| Exp6 | 1.052 | 1.185 | 1.000 | 0.912 | 0.912 | 0.885 | 0.900 | 0.973 |
| Exp7 | 1.060 | 1.235 | 1.193 | 0.971 | 0.902 | 0.959 | 0.978 | 1.036 |
| Exp8 | 1.068 | 1.397 | 1.091 | 1.032 | 0.993 | 0.996 | 1.000 | 1.075 |
| Exp9 | 1.024 | 1.049 | 1.058 | 1.026 | 1.023 | 0.948 | 1.025 | 1.021 |
| Exp10 | 1.201 | 1.035 | 1.087 | 1.101 | 1.103 | 1.088 | 1.090 | 1.100 |
| Exp11 | 1.142 | 1.120 | 1.077 | 1.068 | 1.091 | 1.075 | 1.096 | 1.095 |
| Totals | 1.070 | 1.163 | 1.099 | 1.044 | 1.026 | 1.015 | 1.029 | 1.063 |

Table 6.3: Direct comparison of TIPTOE and LRU-SP/FORESTALL on all two-process experiments and all array sizes. The totals row and column report the geometric mean.

### 6.2.12 Summary of Graphs

Table 6.2 shows, for each array size, the geometric mean of the factor by which TIPTOE performs better than each of the other algorithms, taken over the eleven experiments. Next, Table 6.3 shows a head-to-head comparison of TIPTOE and LRU-SP/FORESTALL. For each experiment described above, and each array size, the table shows the ratio of LRU-SP/FORESTALL's total execution time to TIPTOE's total execution time. At the right/bottom of each row/column the table shows the geometric mean of the values in that row or column. The bottom row is therefore equivalent to the rightmost column of Table 6.2. As the tables show, TIPTOE outperforms LRU-SP/AGGRESSIVE and LRU-SP/FORESTALL by 23% and 16% respectively on 2 disks, when computation and I/O are most closely balanced. On other array sizes, the applications are either compute bound or I/O bound, and the opportunity for strong differentiation of the allocation scheme is smaller. Experiments 4, 5, and 10 show an aggregate improvement of at least 10% for TIPTOE over LRU-SP/FORESTALL across all array sizes, and a substantially better improvement for intermediate array sizes. Experiments 3 and 6 show an advantage of 1% and 2.7% respectively for LRU-SP/FORESTALL.

## 6.3 One I/O-Intensive Process With Background Load

The experiments described above represent traces of pairs of actual applications running side by side. These experiments are central to the thesis because TIPTOE uses load and re-use information to make allocation decisions, choosing how many buffers a particular process may hold at each point. There is a similar situation in which the same tension arises: an I/O-intensive process running on a system with background load. If there is re-use in the background load, the allocator must decide how much of the buffer cache should be dedicated to LRU caching for the background processes, and how much should be given to the I/O-intensive process.

I perform two experiments modeling random and sequential re-use. In some sense, these represent two extremes. A single process performing a cyclic access pattern is difficult to cache for – if fewer buffers are dedicated to caching than the size of the working set then there will be no re-use. On the other hand, re-use according to the traditional LRU hit-rate curve [HP96] is more forgiving — the allocator may dedicate a small fraction of the working set, but still realize a large fraction of potential cache hits. This type of re-use is typically an aggregate over a number of background processes executing simultaneously. I consider each type of re-use in turn.

### 6.3.1 Traditional Background Load

Dahlin [DWAP94] records the activity of an Auspex file server supporting 231 client machines over a one-week period at the University of California at Berkeley.[1] The traces contain around 6.6 million requests and transfer 8.1 Gbyte of read data.

I collected the first 1.09 Gbyte out of 8.1 Gbytes of read requests from the trace set and used this subset to estimate the "hit-rate" profile of the set. To generate the profile, a cache simulation determined for each read where the most recent prior read to the same block occurred, and augmented a counter for that depth. This results in a distribution of re-use: 18% of all accesses are for the immediate prior accesses, 5.6% are for two accesses in the past, 1.5% are for three accesses in the past, and so on. The profile can then be used to generate traces of arbitrary duration and inter-access computation time. I generated traces with a range of durations from fifteen to three hundred seconds, with an expected 15 ms inter-access computation time distributed exponentially. These traces do not provide hints, as they are meant to represent the aggregate background load of a number of processes with small I/O demands.

Figure 6.13 shows the hit-rate profile for the traces. Table 6.4 shows the cache sizes at which the hit-rate crosses 5% intervals. The graph of Figure 6.13 shows that the hit rate

[1] The trace covers three periods of activity: Sept 9–10, Sept 13–15, and Sept 20 – Oct 3.

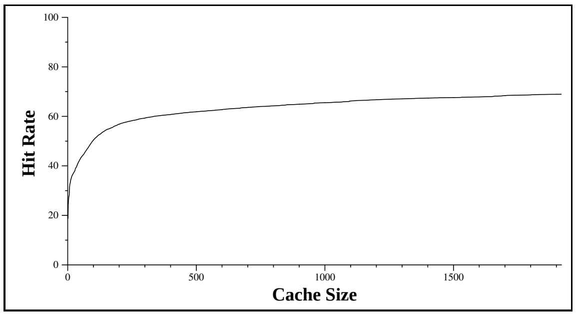



Figure 6.13: Cumulative Hit Rate of Auspex Traces as a Function of Cache Size in Blocks.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 18% | 24% | 31% | 35% | 40% | 45% | 50% | 55% | 60% | 65% | 67% |
| 1 | 2 | 7 | 14 | 36 | 65 | 97 | 162 | 336 | 920 | 1280 |

Table 6.4: Cache Sizes for 5% Increments of Cache Hit Rate

climbs very steeply for small caches, but remains relatively flat for larger caches. Table 6.4 shows that, in fact, the hit rate changes only 12% when the cache size increases from 162 blocks to 1280 blocks. Thus, while this particular background load exhibits substantial re-use (67% for a full cache), most of the caching benefits can be attained with a much smaller cache. Section 6.3.2 considers the opposite situation, in which dedicating too few buffers to the background process will significantly impact the hit rate.

The results are shown in Figure 6.14. As the figure shows, there is no substantial difference between the various algorithms for this workload. However, there are some differences.

On the DAVIDSON trace, LRU-SP/AGGRESSIVE and LRU-SP/FORESTALL perform identically for one disk, since the disk is always constrained. TIP2 performs 1% better than TIPTOE, and 2% better than the LRU-SP algorithms. Although the difference in overall execution time is small, the interaction causing the effect is important so I describe it here in detail. The same effect also arises in the SPHINX/XDS experiment above. As discussed in Section 4.1, TIP2 assigns low cost to evicting items from the hinted cache, and will therefore tend to grow the LRU whenever there is any unhinted re-use. In this situation, TIP2 will cache for the background process, and evict data from DAVIDSON; compared to TIPTOE it incurs 1800 fewer unhinted cache misses, but 11000 fewer hits to the hinted cache.

TIP2's decision to favor the unhinted cache also improves disk performance. Recall that the prioritization scheme assigns high priority to demand reads, and low priority

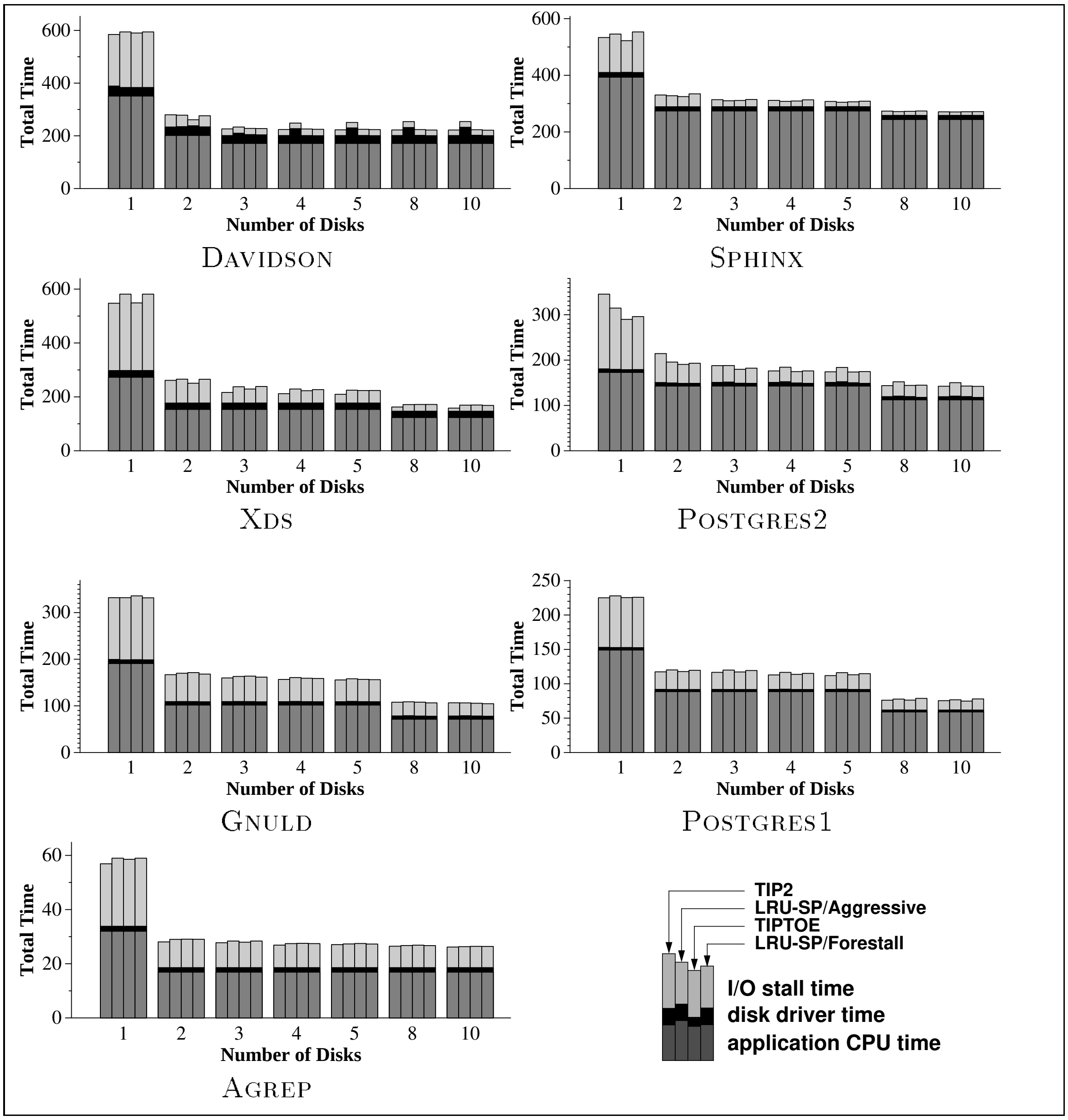



Figure 6.14: Single I/O-Intensive Process with Background Load Profiled from NFS Traces.

both to prefetches, and to prefetches that have become demand reads. Thus, if DAVIDSON is prefetching on a particular disk, whenever the background process wants data from that disk, the head will move as soon as the current transfer completes. This results in many seeks. Extra caching by TIP2 allows more Davidson accesses to be performed simultaneously as the background process hits in the cache, and begins another long period of computation (15 ms on average). This effect results in a 21–25% improvement in average I/O time for TIP2 versus the other algorithms on a single disk, an improvement of 6–21% for larger arrays versus TIPTOE, and a more substantial improvement with respect to LRU-SP/FORESTALL.

The other effect visible in the trace is that TIPTOE performs better than the other algorithms on 2 disks. This occurs because the LRU-SP algorithms dedicate cache to the background process according to its (very slow) rate. Tiptoe dedicates more buffers to LRU caching and generates 8% more unhinted hits than LRU-SP/FORESTALL. TIP2, on the other hand, does not prefetch deeply enough for Davidson. On larger arrays, deep prefetching is no longer required because sufficient bandwidth exists.

For Xds, the COST-BENEFIT algorithms on a single disk dedicate more buffers to caching for the background process since the I/O-intensive Xds process cannot use buffers. The LRU-SP algorithms on the other hand dedicate buffers to Xds because it consumes data quickly. This results in 10% fewer LRU cache misses for COST-BENEFIT than LRU-SP, and a 6% overall improvement in execution time on a single disk. The difference in LRU performance is more dramatic on larger arrays. The LRU-SP algorithms incur 18% more unhinted cache misses than TIPTOE on 3 disks, 15% on 4 disks, 10% on 5 disks, 12% on 8 disks, and 12% on 10 disks. In a sense, Xds is a difficult trace for LRU-SP because it streams through data quickly with little re-use, and therefore collects a large fraction of the cache that would be better used by another, possibly slower, process that exhibits re-use. On larger arrays, however, the additional cache misses have smaller impact on overall execution time because most of the additional I/O can be overlapped against Xds's computation. On 10 disks, for instance, execution time increases by 1% despite 12% more cache misses.

TIP2 and TIPTOE do not perform identically in this experiment. On 3 or more disks, TIP2 outperforms TIPTOE by from 1–7%. On 5 disks, an example with no load-balancing issues, LRU-SP/AGGRESSIVE, LRU-SP/FORESTALL and TIPTOE all complete the Xds trace within $116 \pm 3$ seconds by deep prefetching; TIP2 requires another 34 seconds to complete Xds, but then completes the entire execution 7% faster than TIPTOE.

To understand this behavior note that, ideally, the only time Xds should compute is during I/O for the background process; this would allow maximal overlapping of I/O and computation between the two processes. But TIPTOE and the LRU-SP algorithms prefetch far enough ahead for Xds that it is always ready to compute, so Xds competes with the background process for the processor, and finishes quickly. In general, the typical advantage of unbalanced execution is that the cache can be entirely dedicated to each process in turn. Since Xds has no reuse, and the background process attains almost all its re-use with a small number of cache buffers, this advantage does not apply. Thus, TIP2's slower completion of XDS results in a better overall time.

The SPHINX trace shows TIPTOE executing 6% faster than LRU-SP/FORESTALL on a single disk, and all other timings are less significant. On larger arrays, LRU-SP/AGGRESSIVE performs slightly better than the other algorithms due to SPHINX's late-arriving hints, as described in Section 5.2.4. LRU-SP/FORESTALL on larger arrays typically incurs about 7% more unhinted cache misses than TIPTOE, but overlaps these additional I/O's effectively against SPHINX computation, yielding only small differences in execution time.

POSTGRES1 shows only a tiny advantage to COST-BENEFIT ($< 1\%$) on all array sizes. Note that the decline in overall computation between 1 and 2 disks, and again between 5 and 8 disks, occurs because I use shorter background traces to match the overall execution time of the POSTGRES1 trace as disk bandwidth increases. POSTGRES2 on the other hand shows a small advantage to the FORESTALL algorithms on one to three disks, with TIPTOE performing 1–2% better than LRU-SP/FORESTALL. On a single disk, TIP2 grows the LRU cache but fails to dedicate sufficient resources to caching for POSTGRES2's hinted re-use of inner relation data blocks. (Note that the POSTGRES1 trace does not demonstrate substantial hinted re-use, so this behavior does not occur.) This results in 30% more total I/O's for TIP2 versus TIPTOE, yielding a 22% increase in overall execution time. TIPTOE versus LRU-SP/FORESTALL dedicates more cache buffers to LRU caching of both POSTGRES2's unhinted accesses and the background process. LRU-SP/FORESTALL incurs 6% more cache misses than TIPTOE on a single disk, and 9–16% on larger arrays.

Finally, the GNULD and AGREP cases do not demonstrate interesting effects. AGREP shows a 1–4% improvement for TIP2 on 1–4 disks, and smaller differences for larger arrays. Caching behavior for both hinted and unhinted cache are similar for all algorithms. As in the standalone case, TIP2's marginal improvement on smaller arrays is due to deeper disk queues.

Table 6.5 shows the aggregate results of TIPTOE compared to the other three algorithms. As the table shows there is little overall difference among the algorithms; in fact, on no array size is there more than a 3% impact due to the algorithm.

### 6.3.2 Sequential Background Load

Another common re-use pattern occurs when a process loops cyclically through data. In the section I examine the performance of an I/O-intensive application in the presence of cyclic background load.

I generated traces that cyclically access a working set of 640 blocks, or half the cache (for the single shortest trace, the working set was reduced to 500 in order to demonstrate

| Disks | TIP | LRU-SP/AGG | LRU-SP/Forestall |
|---|---|---|---|
| 1 | 1.021 | 1.029 | 1.020 |
| 2 | 1.026 | 1.024 | 1.022 |
| 3 | 0.993 | 1.020 | 1.012 |
| 4 | 0.987 | 1.029 | 1.006 |
| 5 | 0.986 | 1.028 | 1.002 |
| 8 | 0.988 | 1.029 | 1.002 |
| 10 | 0.990 | 1.029 | 1.002 |
| * | 0.999 | 1.027 | 1.009 |

Table 6.5: Summary of results for traditional background load. This table gives the ratio of the elapsed time for each other algorithm to the elapsed time for TIPTOE. Numbers are the geometric mean of the ratios for the seven experiments considering an I/O-intensive process running on a machine with "traditional" background load as sampled from a large NFS server. The last line, marked with a '*', is the mean over all array sizes.

re-use). Again, the inter-access computation of the background load is exponentially distributed with expected value 15 ms. Traces were created for the same durations as the traditional background profile described above. The results are shown in Figure 6.15; as the figure shows, the difference between COST-BENEFIT and LRU-SP is significant.

I begin again by considering DAVIDSON. On a single disk, TIPTOE outperforms LRU-SP/FORESTALL by 218%. Both of the COST-BENEFIT algorithms require approximately four iterations through the unhinted sequential dataset before the estimators converge to the correct estimate and grow the LRU cache according to that estimate so the entire dataset is cached. The benefit of using those 640 buffers to cache for DAVIDSON is smaller than the benefit of using them for LRU caching. We would expect this to be the case, since each buffer used to cache DAVIDSON data will generate one hit per 2089 accesses, while each buffer used to cache for LRU accesses will generate one hit per 640 accesses. On a single disk TIPTOE takes 2664 misses, while LRU-SP/FORESTALL takes 19,953. TIP2 takes only 1223 misses since it naturally places a low value on caching for hinted re-use, due to its system model, but it completes 6% slower because TIPTOE prefetches more deeply for DAVIDSON.

On larger arrays, on the other hand, all four algorithms perform similarly. This appears counterintuitive: we expect that the LRU-SP algorithms will process the DAVIDSON trace even more quickly when there is extra bandwidth available, but will not be able to process the unhinted trace quickly because there are no hints. In fact, this is exactly what happens. However, processing of the unhinted trace is so dramatically slow that

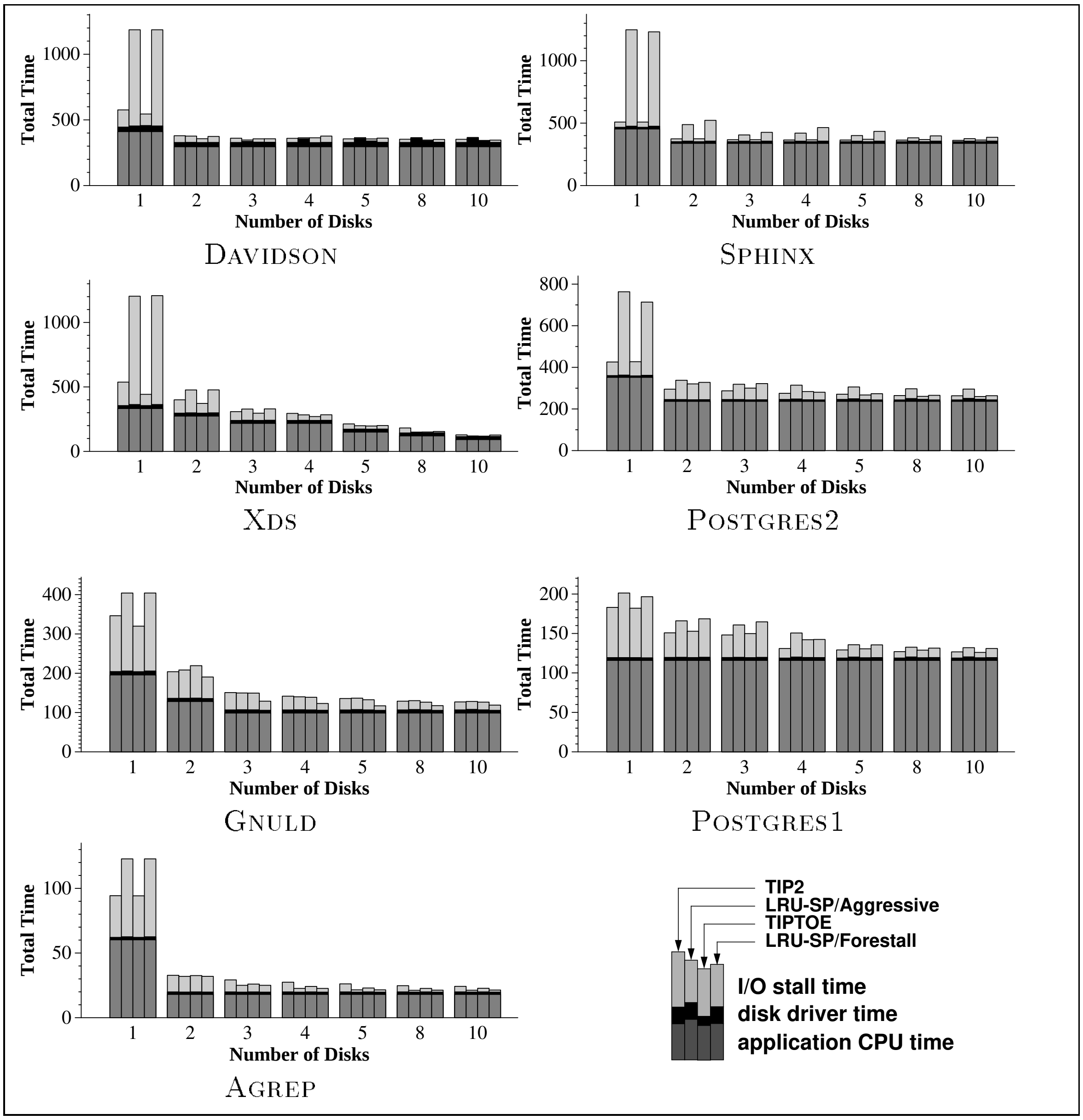



Figure 6.15: Single I/O-Intensive Process with Sequential Background Load.

DAVIDSON actually completes all 130,000 accesses while the background process completes about 5,000 accesses. Subsequently, the background process runs to completion with the entire cache at its disposal.

The same phenomenon occurs on all larger arrays. The anomalous behavior on a single disk occurs for the following reason. On two or more disks, DAVIDSON can always prefetch multiple blocks from at least one disk because the unhinting background load submits only a single access at a time and will restrict its accesses to one disk until it has read an entire 8-block stripe unit. On a single disk, however, DAVIDSON cannot prefetch while the high-priority demand reads from the background process are being serviced. Thus, a read completes for the background process, so DAVIDSON begins a fetch which must do a large seek, and 15 ms later another request arrives from the background process at higher priority than prefetching, drawing the head back to the background process' data space. The average I/O time for the LRU-SP-based algorithms on a single disk is therefore more than 2.5 times as large as TIPTOE's average I/O time.

SPHINX demonstrates a similar pattern, with LRU-SP taking 2.42 times as long to complete on a single disk. I/O times range from TIP2's average 5.62 ms to LRU-SP/FORESTALL's 24.55 ms. On two or more disks, LRU-SP/AGGRESSIVE actually outperforms LRU-SP/FORESTALL because LRU-SP/FORESTALL will occasionally stop prefetching — the window between two unhinted accesses for performing nearly-sequential SPHINX prefetches is therefore not fully exploited, and LRU-SP/FORESTALL's I/O time is 10% higher than LRU-SP/AGGRESSIVE's. LRU-SP/AGGRESSIVE's execution time is 30% longer than TIPTOE's; LRU-SP/FORESTALL's is 40% longer. The pattern continues for larger arrays: LRU-SP/FORESTALL takes 16% longer than TIPTOE on 3 disks, 26% longer on 4 disks, 17% longer on 5 disks, 8% longer on 8 disks, and 6% longer on 10 disks.

The same behavior is visible on XDS. On a single disk, the overall execution time of LRU-SP/FORESTALL is larger than that of TIPTOE by a factor of 2.69, the most dramatic difference of any workload presented so far. With larger arrays, there remains a larger disparity between algorithms than in the DAVIDSON example because XDS has little re-use of its own, and must perform more I/O's with less sequentiality than DAVIDSON. Under LRU-SP, the cache will be dedicated primarily to caching for XDS even though the hints make it clear that there is little re-use. Because the LRU-SP allocator is decoupled from the prefetcher, knowledge about re-use is lost to the high-level allocation algorithm.

On larger arrays, there are two other (related) effects visible in the differences between TIP2 and TIPTOE: load-balancing, and deep prefetching. On 3 disks there are no load-balancing effects, but TIPTOE runs 4% faster because of 2,300 accesses that TIPTOE prefetches completely in time, but TIP2 only partially prefetches before the read arrives. On four disks, load balancing adds to this effect, resulting in 24 seconds difference in overall execution time and 3,800 additional accesses that TIP2 does not complete in time. The same trends are visible on 5 and 8 disks.

POSTGRES2 contains much more internal structure than the other traces. Again we see significant differences between COST-BENEFIT and LRU-SP. LRU-SP/FORESTALL takes 67% longer than TIPTOE on a single disk, but for larger arrays the difference is not substantial.

As a brief digression, this case is interesting because there is contention specifically for the LRU cache: Postgres2 displays significant unhinted re-use, as does the background process. The cost-benefit architecture studied here does not maintain a per-process LRU cache, so any buffers dedicated to unhinted caching will be available to all processes. Thus, the cost-benefit profilers cannot dedicate LRU buffers to caching for Postgres2 alone, or for the background process alone. While maintaining multiple LRU caches might be advantageous in some situations, there would be substantial overhead, and circumstances such as cooperating processes would not be modeled well.

On a single disk under cost-benefit, Postgres2 begins by accessing the outer relation sequentially with hints, and the inner relation index randomly without hints. There is no disclosed re-use to the outer relation, and there are hits to the LRU from inner relation index blocks and from the background process. Thus, cost-benefit grows the LRU at the expense of the posthint cache, and is able to cache the entire working set of the background process along with the inner relation index; both together will entirely fill the cache. At the end of this phase, the LRU contains 1275 blocks. During the next phase, Postgres2 discloses the accesses to the inner relation data blocks. Postgres2 therefore begins to prefetch those blocks, and notes substantial hinted re-use, so there is some splitting of the cache; after about 2000 accesses, the LRU converges to 700 blocks, enough to hold the working set of the background process and the occasional outer relation data block that has been re-read. Overall, TIP2 incurs 3085 misses, tiptoe 3118.

Lru-sp begins by splitting the cache between hinted accesses to the outer relation with no disclosed re-use (although each block will be read exactly once during the remainder of the trace), unhinted accesses to the inner relation, and unhinted accesses to the background process. Reads to the outer relation proceed most quickly, so a substantial chunk of the cache is given to this data. By the end of the first phase, lru-sp/forestall has incurred 2346 misses versus 1914 for tiptoe: not a significant difference. However in the second phase, Postgres2 consumes data at roughly twice the rate of the background process, and therefore garners more than half the cache. The background process derives no re-use from cache buffers, and by the time execution completes, lru-sp/forestall incurs 9686 misses versus 3118 for tiptoe.

On 2 disks, TIP2 outperforms tiptoe by 8% because tiptoe estimates that there is benefit to caching inner relation data blocks for hinted re-use, while TIP2 will essentially grow the LRU whenever there is any benefit to doing so. As Postgres2 is consuming slowly, it takes tiptoe approximately three additional iterations through the background process to estimate that buffers should be given to the LRU rather than the hinted cache, resulting in 2200 additional misses. The same pattern holds for larger arrays.

The behavior of Postgres1 is similar to Postgres2, but is not as extreme because there is less unhinted re-use of the inner relation index blocks, and less hinted re-use of the inner relation data blocks. On a single disk, LRU-SP/FORESTALL takes 8% longer than TIPTOE due to 6.5% fewer hinted cache hits, and as a result shows a slightly higher average I/O time. Similarly on 2 and 3 disks, LRU-SP/FORESTALL takes 10% longer to complete than TIPTOE. On 4 disks, TIP2 performs slightly better than the other algorithms because it completes the background process before the end of POSTGRES1's second phase, and can therefore dedicate the entire cache to inner relation index blocks, resulting in 1200 more hinted cache hits than TIPTOE. On larger arrays, COST-BENEFIT runs from 2–4% faster than LRU-SP, due to 42–50% fewer unhinted cache misses.

Turning to GNULD, on a single disk, LRU-SP/FORESTALL runs for 14% longer than TIPTOE due to 34% more cache misses. On two or more disks, however, LRU-SP/FORESTALL actually runs from 6–16% faster than TIPTOE, a surprising result. The reason is the following. COST-BENEFIT relies on adaptive estimators that respond quickly to new situations. GNULD contains about 4000 unhinted accesses, about half as many as the background process. These unhinted accesses occur primarily in two batches of approximately 2800 and 1400 accesses. TIP2 and TIPTOE use an estimator of the LRU cache hit-rate profile that places heavy confidence on the last 1000 accesses as a good predictor of the future hit rate. When the trace begins and the estimator is being warmed up, an iteration of the background trace completes without any visible re-use. Once the second iteration begins, the estimator begins to see re-use, and quickly increases the cost of taking buffers from the LRU, as we would expect. However, once the batch of 2800 unhinted accesses arrives, the estimator determines that the LRU is no longer valuable and allows buffers to migrate to the hinted cache. When the batch of GNULD unhinted accesses stops, the estimator must see an entire iteration of misses from the background process before it begins to grow the LRU again. This process repeats during the second GNULD batch of unhinted accesses. Thus, on two occasions, TIPTOE must incur one working set worth of misses, totalling 1280 misses, which is approximately the number of additional unhinted cache misses that TIPTOE takes relative to LRU-SP/FORESTALL.

It is important to design an estimator that is correct in as many situations as possible, while recognizing that it is not possible to treat *every* situation correctly. This is the only experiment I have performed in which the cache profiling is misled by unusual application activity, and it does not occur only because a process with phase behavior interacts badly with the load presented by another process.

Finally, on a single disk under AGREP, LRU-SP/FORESTALL takes 30% longer to complete and incurs 104% more cache misses. On larger arrays, LRU-SP performs slightly better than COST-BENEFIT. On a single disk, AGREP takes long enough that the background process can safely perform 6 iterations through its working set. This is sufficient for the TIPTOE estimators to warm up and begin caching effectively. With larger arrays, however, there is only time for two iterations through a 500-block background working set. After the first iteration, there is no indication of re-use, and under TIPTOE all buffers

| Disks | TIP | LRU-SP/AGG | LRU-SP/Forestall |
|---|---|---|---|
| 1 | 1.049 | 1.735 | 1.710 |
| 2 | 0.996 | 1.095 | 1.088 |
| 3 | 1.018 | 1.039 | 1.034 |
| 4 | 1.016 | 1.042 | 1.018 |
| 5 | 1.032 | 1.037 | 1.009 |
| 8 | 1.044 | 1.030 | 1.003 |
| 10 | 1.026 | 1.032 | 1.009 |
| * | 1.026 | 1.124 | 1.104 |

Table 6.6: Summary of results for sequential background load. This table gives the ratio of the elapsed time for each other algorithm to the elapsed time for TIPTOE. Numbers are the geometric mean of the ratios for the seven experiments considering an I/O-intensive process running on a machine with sequential background load. The last line, marked with a '*', is the mean over all array sizes.

---

are considered to be part of the posthint cache. During the second iteration, TIPTOE must decide whether to evict post-consumption buffers from the posthint cache, which has so far exhibited no re-use, or to evict unhinted data buffers from the LRU cache, which has also exhibited no re-use. Arbitrarily, it chooses to evict from the LRU, since it has no information about which decision is correct. During the second iteration, it notices re-use and grows the LRU, but by then the trace completes with no re-use attained. Thus, LRU-SP/FORESTALL performs from 2–6% faster on array sizes from 2 to 10.

This set of graphs represents the other extreme from the NFS-profiled background load described above. In that earlier model of background load, dedicating too few buffers to the background process resulted in only a tiny impact on hit rate; in this model, if one fewer buffer than necessary is given to the background process, there will be no cache hits.

Table 6.6 shows the aggregate results of TIPTOE compared to the other three algorithms. As the table shows there is a substantial difference between COST-BENEFIT and LRU-SP allocation schemes: TIPTOE outperforms LRU-SP/FORESTALL by 71% on average on a single disk, and by a much smaller margin on larger arrays.

## 6.4 Multi-Process Issues

In this section I examine in more detail some issues that arose in the experiments described above. First, Section 6.4.1 compares the hinted cache estimators used by TIP2 and TIPTOE. Next, Section 6.4.2 examines the effectiveness of the posthint cache, as a demonstration of the power of cost-benefit analysis.

### 6.4.1 TIP2 versus TIPTOE

In several of the experiments described above, TIPTOE's prefetching estimator allows it to fetch more deeply and TIP2 when necessary, and consequently to reduce execution time (see, for example, Section 5.2.1 and Section 6.2.4). This section focuses on the corresponding estimator of the cost of evicting hinted data, for TIP2 and TIPTOE. In general, as discussed in Section 4.1, TIP2 will give cache buffers to any unhinting process that will use them, even if a hinting process would like to hold the buffers for hinted re-use, since TIP2 assumes that the hinted process will be able to prefetch the missing blocks without stall. TIPTOE, on the other hand, makes a more careful estimate of the benefit of caching for hinted versus unhinted re-use.

It makes sense to ask whether TIP2 will always perform roughly as well as TIPTOE, to understand whether the additional complexity of the TIPTOE estimator is worthwhile. In the section, I set aside the LRU-SP algorithms for the moment, and induce tension between hinted and unhinted re-use to compare the two COST-BENEFIT algorithms.

On traces running with synthetic unhinted background loads, as in Section 6.3, both TIP2 and TIPTOE dedicate sufficient buffers to caching for the unhinting process, and the hinted cache estimator is not necessary. In this section, I consider the opposite situation in which there is some unhinted re-use, but the buffers are more effectively used for hinted caching.

I assume that a fast, I/O-intensive, hinting process accesses data according to a strided pattern (every $4^{th}$ block), with a working set of 1,000 blocks, and a 1 ms average inter-access computation time. The process streams through the dataset 60 times. In the background, a slower process reads blocks at random for the majority of accesses (set at 90%), and reads the block it read 1,200 accesses in the past the remaining 10%. Thus, the slower process has some re-use, but even with a large cache dedicated to it, few accesses will hit in the cache. I consider slower processes that read 6,000, 8,000, 10,000, 12,000 and 14,000 blocks, for a single-disk array. The results are shown in Figure 6.16.

In general, TIPTOE outperforms TIP2 by 34–40% over the different durations of the background process. TIP2 and TIPTOE both profile the LRU effectively and estimate the cost of taking buffers from the LRU. Since there is a small hit rate for large caches,

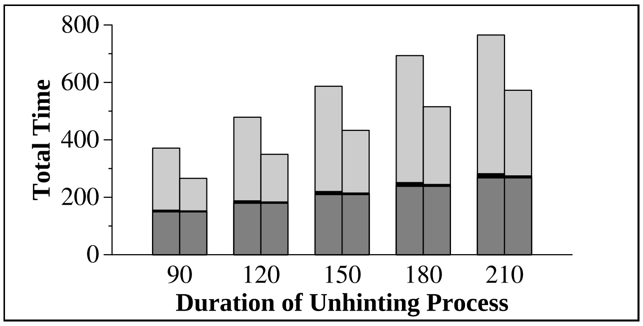



Figure 6.16: TIP2 and TIPTOE make allocation decisions for pairs of traces. The first trace requires hinted caching; the second requires unhinted caching. The $x$ axis shows the duration of the unhinting process. Once the unhinting process runs for long enough for TIP2's estimator to warm up, TIP2 dedicates buffers to LRU caching when the same buffers could be used more beneficially for hinted caching. TIPTOE's more effective estimator of the benefit of hinted caching successfully determines the correct allocation.

both algorithms present some small cost to the allocator. TIP2 then estimates the cost of taking a buffer from the hinted cache according to Equation 4.1, and gives the buffer a low eviction cost under the assumption that it can be read back without stall. This assumption is violated, since TIP2's model of sufficient disk parallelism does not hold on these traces in the single-disk case. TIPTOE's estimator notes that the disk is constrained, and that if the block is evicted, it will need to be re-fetched, incurring stall. Thus, TIPTOE dedicates more buffers to hinted caching than TIP2.

With a 60-second background process, TIPTOE caches the hinted working set of the fast process, allowing that process to complete quickly, and then continues to service the background process. TIP2 grows the LRU cache to 1200 buffers, and generates only 8% re-use for the fast process while the background process continues to run, reading according to the hints, and then evicting the data despite known re-use. TIP2 performs 57% more I/O than does TIPTOE on this trace. A similar pattern holds for the other instances. With a 210-second background trace, for instance, TIP2 performs 88% more I/O (by time) than TIPTOE, and takes 34% longer to complete.

### 6.4.2 Post-Consumption Hints

When a hinted read arrives and there is no future hint for the same data, the system must decide how long the block should be kept in memory. In the original TIP2 system the block was added to the tail of the LRU queue under the assumption that, since it was recently accessed, it might be accessed again in the near future. However, if an XDS-like process is streaming through a large amount of hinted data with minimal re-use, and another process like POSTGRES2 is performing unhinted reads with strong locality[2] this policy will "dilute" the LRU queue with buffers that are never re-used. The opposite policy is to take lack of hints for a block as a "release" of the block, and place the block on the head of the LRU queue for immediate eviction. However under this policy a process such as SPHINX, which offers hints in small batches just before the data is required, would be prone to flush blocks that might soon be hinted.

As described in Section 4.8.10, the cost-benefit framework provides a simple, elegant solution to this problem. Rather than releasing these problematic "posthint" buffers to the LRU queue, the system instead releases them to a separate posthint queue which maintains an independent estimate of the value of its buffers. If the posthint buffers are often re-read, as in SPHINX's case, the allocator will choose to grow the posthint cache at the expense of the LRU cache. On the other hand if the posthint buffers are never accessed but unhinted accesses demonstrate re-use, as in the case of POSTGRES2 and XDS, the allocator will instead choose to dedicate resources to the LRU cache.

In general, the cost-benefit framework allows the system designer to identify subclasses of a resource that display uniform or similar patterns of behavior or re-use. The designer can then tailor estimators to each subclass, such as posthint buffers or unhinted buffers. Buffers can be members of multiple classes, and can be valued by multiple estimators,[3] and the allocator will automatically incorporate any new estimates into the global valuation described earlier.

This section performs two experiments to study the posthint estimator alongside the two "static" policies of always releasing post-consumption buffers either to the head or to the tail of the LRU queue. The conclusion is that there are situations in which these static policies can perform substantially worse than an adaptive posthint estimator, which tracks the best performance of the two static policies. In the first experiment, a non-hinting process with re-use is running alongside a hinting process with no re-use, much like the XDS example described above. Post-consumption buffers should not be cached since the hinting process never re-uses these buffers. On the other hand, the non-hinting process needs cache buffers. Therefore, we would expect that releasing post-consumption buffers to the tail of the LRU would be a poor policy in this case, while releasing them to the head of the LRU to be evicted immediately should perform well. In more detail,

[2]The same situation could arise within a single process if the process hints a large fraction of its accesses, but is unable to hint another set of accesses with higher re-use.

[3]For instance, a block could be read as a demand miss, placed into the LRU queue, and then have a hint arrive that says it will be read again in 300 accesses. The LRU estimator and the hinted cache estimator will independently value the buffer, and if either estimator assigns high cost the buffer will not be evicted.

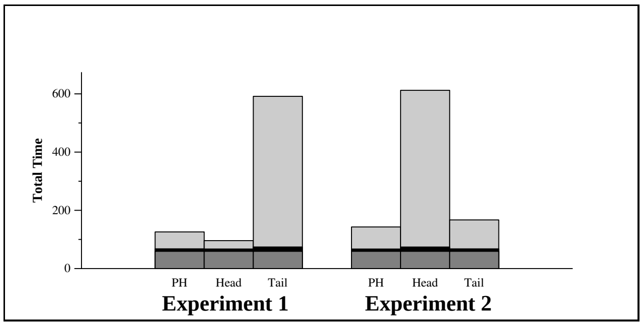



Figure 6.17: TIPTOE performs two experiments under three different policies for post-consumption hinted data. The first policy, "PH,", estimates the re-use opportunities for this type of data, and uses the estimate to make caching decisions. The second policy, "Head," always releases post-consumption blocks to the head of the LRU. The third policy, "Tail," always releases post-consumption blocks to the tail of the LRU. In Experiment 1, post-consumption data will never be re-used, and under the Tail policy it dilutes good data in the LRU and causes an increase in I/O. In Experiment 2, traditional LRU data is not re-used but post-consumption blocks are re-used. Under the Head policy post-consumption data is evicted before it can be re-used, resulting in a substantial increase in I/O.

the first process performs 15 cycles through a 1000-block dataset without hints, and the second process performs 1 cycle through a 15000-block dataset with hints.

In the second experiment, a non-hinting process without re-use runs alongside a looping process with posthint re-use. In this case, the non-hinting process has no use for cache buffers, but the hinting process would like to retain its data for later re-use, after another batch of hints arrives. We expect that a policy of releasing post-consumption buffers to the tail of the LRU will allow the hinting process to re-read its buffers from cache, and will therefore perform well, while a policy of releasing to the head of the LRU will cause the hinting process to re-read substantial amounts of data. The first process accesses 15000 blocks sequentially without re-use, and the second process performs 15 iterations of hinting and then consuming the same 1000 blocks. The results of both experiments are shown in Figure 6.17.

## 6.5 The Multi-Process Case: Lessons Learned

We have distilled our experience with informed caching and prefetching into ten lessons which these experimental results illustrate. The primary lesson is that LRU-SP induces a partition of the buffer cache that, to first order, depends on the relative access rates of the processes, but that relative access rate is not a good predictor of caching value. In contrast, COST-BENEFIT partitions the cache based on estimates of the value of each piece of data. This algorithmic difference is responsible for the greatest performance differences in the graphs, and was the initial reason we undertook this study. Other lessons have significant impact, but often in particular situations such as when a disk is routinely constrained or unconstrained. Throughout the discussion we refer to experiments by the experiment numbers in Figures 6.1–6.12.

**Lesson 1:** *Access rate is not a good predictor of caching value.* The experiments reveal two situations in which differences between data rates and re-use characteristics result in LRU-SP and COST-BENEFIT finding different allocations. First, if a trace displays a substantial fraction of unhinted reads but consumes data slowly, LRU-SP will not dedicate buffers to LRU caching and will suffer demand misses. COST-BENEFIT's LRU estimator will detect re-use and publish a high value, reflecting the advantage of using buffers for LRU caching, and the COST-BENEFIT allocator will grow the LRU cache in response. Second, if one hinting process shows significant re-use and another hinting process shows little re-use, but their relative access rates are not similarly disproportionate, the hinted cache of the first process will be larger under COST-BENEFIT than under LRU-SP. COST-BENEFIT will send the post-consumption blocks of the low re-use process to the LRU queue where estimation will show that they are not used and can be evicted with low cost;[4] it will assign a higher value to the hinted blocks of the high re-use process, so will allow the high re-use process a larger fraction of the cache. LRU-SP will partition the cache based on process rates, giving a large fraction to the low re-use process.

The first situation, in which a process exhibits a large number of unhinted reads but does not consume data quickly, appears in sequential background experiments for XDS, SPHINX, AGREP and POSTGRES2, and in two-process experiments 4, 5, 6, 7, 10, and 11 (most notably experiment 4). The POSTGRES1 and POSTGRES2 traces, which issue large numbers of unhinted accesses, appear in all these experiments. These traces also include a phase of unhinted accesses followed by a phase of hinted accesses so the cache management algorithm must also be adaptive to this change in application behavior. In experiment 4, for instance, LRU-SP/AGGRESSIVE suffers a factor of 6.7 more demand misses than TIPTOE on two disks, and still shows 2.6 times as many demand reads on ten disks even though sufficient bandwidth exists to prefetch all hinted data without stall.

Experiment 1 demonstrates the second situation described above, in which two hinting processes compete for buffers but one process has more re-use and therefore uses buffers

[4] Section 6.4.2 shows that, if these post-consumption blocks exhibit different re-use patterns than unhinted demand read blocks, it is simple to integrate a separate "posthint estimator" into the COST-BENEFIT framework.

more effectively. Average computation time per operation is 0.83 ms for DAVIDSON and 0.72 ms for XDS. However, when each process runs alone XDS re-uses only 3.5% of its data while DAVIDSON's re-use is 38%. On two disks, LRU-SP/AGGRESSIVE shows 43% fewer hinted cache hits than TIPTOE because it dedicates buffers to holding XDS data blocks that will not be re-used.

**Lesson 2:** *On unconstrained disks, hinted blocks can always be prefetched in time so caching them is not as important as caching for unhinted accesses.* The background-load experiments involving POSTGRES1 and POSTGRES2, under either traditional or sequential load, and Experiments 4, 5, 6, 7, 10, and 11, all show an increase in stall for LRU-SP/AGGRESSIVE and, in some instances, LRU-SP/FORESTALL on larger arrays. None of this stall results from hinted reads that have not completed; it all results from unhinted reads, some of which could have been cached if sufficient buffer resources had been given to the LRU queue. In experiment 6 on ten disks, for instance, LRU-SP/AGGRESSIVE incurs 2.2 times as many demand misses as TIPTOE: 4303 versus 1851. These misses translate directly into stall if they cannot be overlapped against computation in another process.

**Lesson 3:** *On constrained disks, hinted blocks that are ejected cannot be re-fetched without stall so caching them is as important as caching for unhinted accesses.* TIP2's hinted cache estimator assumes disks are unconstrained and estimates that ejecting a hinted block beyond the prefetch horizon will only add $T_{\text{driver}}$ overhead. TIPTOE modifies this estimator so that if the block lies on a constrained disk, the cost of ejecting the block includes the stall to re-fetch it (see Equation 4.4). This effect is significant in experiments 1, 2 and 8, and when POSTGRES1 runs under profiled background load. In experiment 8 on a single disk, for instance, TIPTOE caches 37% more data blocks for DAVIDSON despite TIP2's much more conservative prefetching policy because TIPTOE values the blocks more highly.

**Lesson 4:** *Eviction decisions impact locality of "re-fetched" data.* Hinted cache data that is evicted must be fetched back later. A prefetching scheme may attempt to select data for eviction so as to increase disk locality when the data must be read back in. As mentioned above, a full treatment of this topic requires a theoretical model of non-constant disk service time so a treatment within TIPTOE is beyond my scope; nonetheless, the phenomenon arises in simulations. As described in Section 4.5, Cao et al in their presentation of AGGRESSIVE [CFKL95b] describe a mechanism they call "batching" in which the prefetching algorithm waits for the disk to go idle and then submits up to $B$ requests, where the batchsize $B$ is a parameter of the algorithm. LRU-SP, TIPTOE and LRU-SP/FORESTALL all adopt this scheme in my implementation. On cyclic datasets, the $B$ evicted elements are typically the most recently consumed blocks. Since neighboring blocks in the access stream display locality on the disk, this scheme allows the blocks to be refetched with low average disk service time. In experiment 2, for instance, TIP2's average I/O service time is 7% larger than the other algorithms on a single disk. The difference is not even larger because TIP2's conservative prefetching does not evict many blocks from the hinted cache. However, LRU-SP/AGGRESSIVE's average disk service time increases by 15% in experiment 1 when its batching mechanism is turned off and it instead submits its prefetches to the driver one at a time.

**Lesson 5:** *Constraint-aware prefetching only reasons about* known *constraints.* The SPHINX trace typically gives small batches of hints. There are 113 batches containing 449 hints as the program reads a large dictionary file at the beginning of the trace, then 50 other batches containing more than 64 hints, and the remainder of the batches (41% of the total batches) follow the distribution shown in Figure 6.10. Experiments 2 and 9 show LRU-SP/AGGRESSIVE exhibiting less stall than TIPTOE and LRU-SP/FORESTALL on various array sizes because the FORESTALL algorithms, noting that the small batch of hints received so far do not require prefetching, assume that future batches will not cause the disk to become constrained. LRU-SP/AGGRESSIVE, on the other hand, begins prefetching immediately. In experiment 9 with ten disks, for instance, TIPTOE and LRU-SP/FORESTALL incur 1.55 and 2.05 times as many prefetches that have not completed when the read arrives as LRU-SP/AGGRESSIVE does.

**Lesson 6:** *Deeper disk queues yield lower average disk service times.* Both TIP2 and TIPTOE are based upon a system model that assumes a constant disk service time, so modeling of queue sorting and locality is beyond the scope of our theoretical analysis. However our simulator performs CSCAN sorting in the queues and our disk simulator includes such non-constant effects as seek, rotate and transfer latencies, SCSI bus overhead and on-disk readahead buffering. Therefore we exhibit effects resulting from the policy used by each prefetching algorithm to determine when exactly to submit prefetches to the disk driver. TIP2's policy is to submit prefetches out to the prefetch horizon, which in our implementation is 68; thus, TIP2 will commonly keep 68 buffers at the disk queue. The other algorithms submit up to sixteen requests whenever the disk goes idle in order to attain the benefits discussed in Lesson 4, but in doing so they typically generate shorter disk queues. Experiments 5, 7 and 9, and AGREP under profiled background load, display this effect. In experiment 5, for instance, TIP2's average disk service time is 18% faster than TIPTOE's on a single disk. Furthermore, experiment 9 shows that queue depth can interact with stripe unit size to give non-intuitive results — sometimes adding disks can actually decrease the average I/O time.

**Lesson 7:** *Leaving a constrained disk idle leads to additional stall.* This effect was documented in [KTP+96] for the single-process case. We discuss it in Figure 4.3, and mention it here because it arises in the two-process case as well. Experiments 2 and 8 both show TIP2 performing worse than TIPTOE on a single disk; in experiment 2, for instance, on a single disk TIP2 has 7% more prefetches still in progress when the corresponding read arrives than TIPTOE because TIPTOE is willing to perform deep prefetches when the disk goes idle. This effect alone is not responsible for all the difference between the two algorithms in these experiments; Lesson 3 is the primary contributor to the disparity.

**Lesson 8:** *Submitting an I/O requires $T_{\text{driver}}$ computational overhead.* This effect was also documented in the single-process case in [KTP+96]. We discuss it in Figure 4.4, and it appears in experiments 1, 2, 5, 8 and 9 on larger array sizes. In experiment 1, for instance, LRU-SP/AGGRESSIVE on ten disks incurs 52% more driver overhead than TIPTOE.

**Lesson 9:** *Over-aggressive prefetching may result in eviction of prefetched but unread data.* LRU-SP/AGGRESSIVE prefetches deeply even when no disk is constrained. If a prefetching process is running alongside either another prefetching process or a process with significant demand reads, the prefetching process might fetch a block and then be asked to give it up to provide a block to the other process. Experiments 1, 4, 8 and 9 display this behavior; experiment 8 is the most consistent example with LRU-SP/AGGRESSIVE on five disks evicting 26% of its data before reading it. These evictions do not increase stall; in fact, LRU-SP/AGGRESSIVE stalls less waiting for prefetches to complete than any other algorithm. However, again on five disks, LRU-SP/AGGRESSIVE incurs 80% more $T_{\text{driver}}$ overhead than TIPTOE, adding 13% to the overall execution time. The other instances within this trace, and in the other specified experiments, are less significant.

**Lesson 10:** *Batching for efficient I/O may be defeated by prioritization schemes.* In Experiment 3 and DAVIDSON under both profiled and sequential background load, priority scheduling causes the disk head to seek back and forth between the data of two different processes, reading only short amounts of data for each process, even though one process has deep, correct hints.

# Part II

# Theory

# Chapter 7

# Theory Overview

*But stop!–these theoretic fancies jar on serious minds*

*— William Wordsworth, "The Excursion"*

Part II of the thesis considers a class of problems called *online problems*, for which it is necessary to give the solution incrementally, committing to each part of the answer before the next part of the problem becomes available. The online algorithms field has developed a set of standard notations, techniques, folklore results and so on, that often appear without introduction and make for dense reading. This chapter is meant to be both a high-level description of the results to follow, and an introduction to the field so the document will stand alone. Sections 7.1, 7.2, and 7.3 all contain background material. Section 7.4 describes the particular problem I study and the results I attain. Section 7.5 describes a modification to the traditional online model, and describes some results for the new model that draw heavily on the algorithms of Section 7.4. Finally, Section 7.6 discusses related work in the field, with a focus on results that either led to these results, stemmed from these results, or share techniques.

The results in this part of the thesis are all collaborative. I will describe details of the collaboration in each chapter, but in general, Avrim Blum contributed to all the topics described here, Merrick Furst contributed to the results of Chapter 9, and the results of Chapter 8 were later extended (substantially) by Yair Bartal, Avrim Blum, Carl Burch and myself [BBBT96], improving the proofs and presentation in this document.

## 7.1 Online Problems

*Online algorithms* must process a sequence of requests, making decisions about early requests before seeing later requests. Informally, we face on-line problems routinely. For instance, when driving on the highway, we must choose a lane. At each point in time we may stay in our current lane or, with some effort, switch lanes. The central problem here is the same as the central problem in the formal version of the problem: if we switch lanes because the other lane is moving faster, our new lane will immediately slow down according to Murphy's law and we will have made the wrong decision because we don't know the future. The goal of a good online algorithm in the metric we adopt is, informally, to perform as well as possible whatever the future may be.

At heart, a traditional algorithm takes some input and provides an output. An online algorithm, on the other hand, must respond to an entire sequence of requests, and the response to any particular request must not depend on any later request. In general, an *online algorithm* $A$ for a request sequence $\sigma = \sigma_1, \sigma_2, \ldots$ is an algorithm whose response to $\sigma_i$ may not depend on $\sigma_j$ for any $j > i$. $A$'s response to each element of the sequence will be drawn from some set of possible responses dependent on the particular online problem. The problem will also specify which responses are acceptable at each point. Each possible response of the algorithm will have some cost associated with it. In the driving example presented above, the elements of the sequence could be the current speeds of traffic in each lane, and the possible decisions of the algorithm would be to stay put or to switch into another lane. Section 7.3 formally presents the $k$-server problems and the Metrical Task Systems, two common classes of online problems that refine the definition above; all the results in this part of the thesis concern these two classes.

## 7.2 Competitive Analysis: A Metric for Online Problems

In this section I describe *Competitive Analysis* [ST85, KMRS88], a metric developed by Sleator and Tarjan twelve years ago that has become successful for analyzing online problems. Let $\mathrm{OPT}(\sigma)$, the *optimal offline cost of sequence* $\sigma$, be the lowest cost attainable by any algorithm on sequence $\sigma$, including algorithms that know the entire sequence in advance. We say that algorithm $A$ achieves competitive ratio $r$ (or "is $r$-competitive") if for some constant $c$, and all sequences $\sigma$, we have: $A(\sigma) \leq r\mathrm{OPT}(\sigma) + c$. Thus, an $r$-competitive algorithm must always perform within a factor $r$ of optimal, plus a fixed additive constant.[1]

The competitive ratio of a problem is defined to be the minimum over all algorithms for the problem of the competitive ratio of the algorithm. These definitions are due to Karlin, Manasse, Rudolph and Sleator [KMRS88]. In other work, the competitive ratio

[1] This is sometimes referred to as *weak competitiveness*, distinguished from *strong competitiveness* in which $c = 0$.

is sometimes referred to as the *competitive factor*, and is related to the *regret ratio* of theoretical finance.

These definitions can all be extended to randomized algorithms in a straightforward manner. The cost of a randomized algorithm on a particular sequence is replaced by the expectation over the coin flips of the algorithm of the cost on the sequence. The optimal cost remains the same, and the competitive ratio is defined analogously using the expected cost of the algorithm. We are concerned primarily with randomized algorithms.

Since the competitive ratio is defined as a worst-case over all sequences,[2] it is often convenient to think of an adversary responsible for choosing the worst sequence. And since the worst sequence for one algorithm may be different from the worst sequence for another algorithm, we think of the adversary as choosing the bad sequence based on the particular algorithm, or in other words, having access to the code of the algorithm. There are two commonly-studied adversary models: the *oblivious* adversary and the *adaptive* adversary. The former must commit to an entire request sequence in advance, while the latter may view the algorithm's decision on each input before deciding on the next element of the sequence. The definition of competitive ratio as given above is a maximum over sequences, and therefore corresopnds to the oblivious adversary model. In the deterministic case, note that the two models are equivalent — the oblivious adversary has access to the code of the algorithm and may therefore build a sequence knowing how the algorithm will react to each element. In the randomized case, however, the algorithm may flip coins as it proceeds. The adversary will know when the algorithm chooses to flip a coin, but will not know the outcome. So we view the adversary as knowing the *distribution* of the algorithm's responses, but not the actual responses themselves. It is common to study randomized algorithms in the oblivious adversary model because the benefit randomization provides is limited against an adaptive adversary. Ben-David *et al.* [BBK+90] showed that a $c$-competitive randomized algorithm against an adaptive adversary implies a $c$-competitive deterministic algorithm. In fact, they show that even if the adaptive adversary is weakened so that, as it generates the next request, it must also serve it (called the *adaptive online adversary*), a $c$-competitive randomized algorithm implies a $c^2$-competitive deterministic algorithm. All the results below are in the oblivious adversary model.

## 7.3 Sub-Classes of Online Problems

The general definition of online problems given in Section 7.1 has been refined into a number of specific classes of problems. The work in this thesis addresses two such classes: the *$k$-server problems* (or simply *server problems*) and the *metrical task systems*.

[2]For competitive algorithms, Murphy's law is a correct and complete representation of reality.

### 7.3.1 $k$-Server Problems

The *k-server problems* were introduced by Manasse, McGeoch and Sleator [MMS88b, MMS90]. A server problem is defined by an $n$-vertex graph with a distance $d$ on the vertices. There are $k$ *servers* inhabiting vertices of the graph. A request is simply a vertex, and represents the requirement that a server be moved to that vertex. If a server is already present there is no cost. Otherwise, a server must be moved, at cost equal to the distance from the original location of the server to the request. The cost to an algorithm on a sequence is the sum of the costs of all server movements required to process the sequence. Given this cost function, the competitive ratio of an algorithm or a problem is then defined as in Section 7.2.

A number of particular $k$-server problems have been studied in depth; I include some quick examples here. Section 7.6 describes the source and current best-known results for all of these problems.

#### Example 1: Caching

A $k$-server problem is characterized by the underlying metric space, and the most natural space is the *uniform space* in which the distance between every pair of points is 1. This problem corresponds to another common computer science problem: cache management. More formally, the cache management problem is the following:

**The cache-management problem:** Given a main memory of $n$ elements and a cache of $k$ elements, service a sequence of requests in an online manner. Each cache element may hold any element of main memory. Requests for memory elements that reside in the cache incur no cost. Requests for elements that are not in cache must be loaded into the cache, evicting some cached element to make room. The cost of servicing a sequence of requests is the number of accesses to main memory.

The correspondence between the cache management problem and the $k$-server problem on the uniform space is the following. The nodes of the graph correspond to the elements of main memory, and the servers correspond to cache elements. Whenever a server occupies a node of the graph, we consider that element to the cached. When a request arrives for a node with no server, some server must be moved to that node, corresponding to an eviction of the vertex currently occupied by the server and an access to main memory (at cost 1) to load the requested element.

**Example 2: Seek-Minimizing Algorithms for Multi-Head Single-Platter Disk Drives**

Consider a somewhat more complicated example in which multiple independent disk heads travel back and forth across the surface of a disk platter. In this problem the cost to service a particular request is the cost to move the head to the appropriate cylinder, and we assume further that the heads move linearly between the outermost and innermost cylinders. Let $n$ be the number of cylinders. The corresponding metric space is given by $n$ equally-spaced points on a line, where the distance between two points is the distance along the line. Servers correspond to disk heads, and moving a head from one cylinder to another corresponds to moving a server from one point to another. Since the metric space looks like a line of points, the problem is commonly referred to as the $k$-server problem on a *line*.

**Example 3: The Spin-Block Problem**

The problem of whether to rent or buy skis for each ski trip is a recurring example in the online community. The model is the following: on each trip the algorithm must either rent skis for $1 or buy skis for some large fixed cost. How many times should the algorithm rent before buying? Formally, the problem has been studied in the context of process scheduling, in which a process waiting on a lock may spin, incurring cost proportional to the waiting time, or block, incurring a single large context switch cost. How long should the process spin before it should be swapped out? This is simply the continuous version of the ski-buying problem.

### 7.3.2 Metrical Task Systems

*Metrical task systems* were initially described by Borodin, Linial and Saks [BLS87]. An MTS is presented as a graph of $k+1$ points, called *states*, and a metric on the states. At any point, the algorithm must be in exactly one state. A request $\sigma_i$ is a vector of $k+1$ values, $\sigma_i^{(1)} \ldots \sigma_i^{(k+1)}$. Upon receiving a request, the algorithm first decides in which state to service the request (say, state $j$). It pays the distance between its current state and state $j$ (typically we think of distances as being large compared to the elements of $\sigma_i$), plus the cost of servicing the request in state $j$, $\sigma_i^{(j)}$.

As an example of a metrical task system, consider the problem of data structure management faced by compiler writers. A particular data structure, such as a matrix, may be represented in memory in a number of different layouts, corresponding to the states of the task system. The metric on the states represents the cost of converting from one layout to another. Operations performed on the matrix will have different costs depending on the layout, or state. The elements of $\sigma_i$ represent the cost of operation $i$ in each state — some operations will be cheap and some expensive for each particular layout. As the sequence of operations arrives, the application would like to modify the data structure dynamically to find an appropriate layout for the types of operations that occur in the sequence, *no matter that the sequence might be.*

I now present two "folklore theorems" that have not appeared formally but that are known in the online algorithms community; one is a useful tool restricting the nature of sequences that need be considered in developing MTS algorithms, and the other connects the MTS problem to the $k$-server problem on $(k+1)$-point spaces. For completeness, I include proofs of these results in an appendix. We say that a task vector is *elementary* if it has only one non-zero element. It is $\epsilon$-*bounded* if every element lies in the range $[0..\epsilon]$.

**Theorem 3 (folklore)** *For any metric space and any fixed $\epsilon > 0$, given an $r$-competitive algorithm for the metrical task system problem with $\epsilon$-bounded, elementary task vectors, it is possible to construct an algorithm for the general metrical task system problem with competitive ratio $(1+\epsilon)r$.*

**Proof:** See Appendix A

**Theorem 4** *Consider a metrical task system on a metric space $M$ of $k+1$ points, and the corresponding $k$-server problem on a $(k+1)$-point space. Given an algorithm $A$ that solves the MTS with competitive ratio $r$, and any positive $\epsilon$, it is possible to construct an algorithm $B$ for the $k$-server problem with competitive ratio $r$. Likewise, given an algorithm $B$ with competitive ratio $r$ for the $k$-server problem, it is possible to construct an algorithm $A$ with competitive ratio $7r$ for the MTS.*

**Proof:** See Appendix A

## 7.4 Weighted Caching

Chapter 8 considers *weighted caching*, an extension to the traditional caching problem described above with connections to the $k$-server and metrical task system models.

A *weighted-cache space* is defined to be a set of $n$ points and a real value corresponding to each point called the *weight* of the point. The distance between any 2 points $i$ and $j$ is defined to be the weight of $j$, regardless of the value of $i$: $d_{ij} = w_j$.

For convenience, we may think of the opposite version of the problem in which the distance from $i$ to $j$ is the weight of the source, rather than the weight of the destination: $d_{ij} = w_i$. Or equivalently, we may consider the symmetric version in which the distance is the average of the weights of the source and destination: $d_{ij} = 1/2(w_i + w_j)$. The equivalence results from the following observation. Imagine a long sequence $\sigma$ of requests. Each server managed by an algorithm will traverse some path through the metric space over the course of processing $\sigma$. In the original formulation of weighted cache spaces, the algorithm will pay the weight of a point upon entering the point. In the new formulations above, the algorithm will pay upon departing, or will pay half upon entering and half upon departing. The sum of all costs incurred by a particular server over the course of processing $\sigma$ will therefore consist of the weight of all intermediate points the server passes through, plus the weight of either the initial point, the final point, or the average of the two. Thus, the total cost to the algorithm will be equivalent under all three measures to within a factor equal to the diameter of the space times the number of servers, independent of the length or cost of $\sigma$. Thus, since the definition of competitive ratio allows a constant factor independent of $\sigma$, any algorithm competitive under one distance will also be competitive under the others.

As a practical application of weighted caching, consider a web browser that may store some fixed number of pages (for instance, assume that image loading is turned off so all the pages are text-only, and therefore roughly the same size). The goal of a cache management algorithm is to reduce the amount of time the user spends waiting for a page to arrive. Thus, the traditional LRU page replacement policy might not be the best policy here. If the LRU page comes from a very distant server, it might be worth keeping it in the cache for longer in case the user requests it again.

This problem can be formulated as weighted caching in the following manner. The "weight" of a page is the time it takes to read the page from its web server. The $k$ servers correspond to the $k$ pages that may be in the cache at any given time. We adopt the formulation $d_{ij} = w_j$ so that whenever a server moves from point $i$ to point $j$ (*i.e.*, whenever the cache management algorithm evicts page $i$ and loads page $j$) the cost to the algorithm is the weight of page $j$, or the time to load the new page. The total cost of servicing a sequence of requests under the metric will be the amount of time the user must spend waiting for pages to arrive, as desired.

The primary result of Chapter 8 is an $O(\log^2 k)$-competitive randomized algorithm for weighted caching on $(k+1)$-point spaces (also known Cat-and-Mouse problems or Pursuit-Evasion games [BKRS92]). We also give an $\Omega(\log k)$ lower bound on the competitive ratio of any such algorithm for every fixed weighted-cache space, extending and simplifying results of [KRR91].

By theorem 4, we also have an $O(\log^2 k)$-competitive algorithm for any weighted caching task system, and an $\Omega(\log k)$ lower bound for any weighted caching task system.

## 7.5 Free Time

Next, in Chapter 9 we consider a model of *free time* whose internal structure is closely related to weighted caching. In the traditional on-line model, an algorithm is asked to process a request sequence. Each request is presented after the algorithm has completed processing the previous one and the cost of the algorithm is the cumulative time or work needed. In many natural on-line settings, however, requests may arrive infrequently relative to the speed of the algorithm. In these situations, it makes sense to model an algorithm as having free time, for which it is not charged, between the servicing of one request and the arrival of the next. For instance, a classical example on-line problem is the "servers are fire trucks" problem in which requests represent fires, and when a request arrives some server, or fire truck, must be moved to the fire as quickly as possible. In this example, one rightly cares much more about the time it takes to get a fire truck to a fire once a call has been made and cares much less about any time spent moving trucks to resting places while there are no fires to attend to.

Similarly, consider again our earlier example of driving on a multi-lane highway. In some situations traffic is tight and the traditional model is an appropriate abstraction of the situation. However, in other situations, there are lulls in the traffic in which it is possible to change lanes for free. In these situations, free-time might be a better abstraction.

In general, in computing situations, if the process issuing requests is substantially slower than the process serving requests then the server is liable to have a fair amount of free time at its disposal between demands. In these situations it makes sense for the server algorithm to use the free time between requests to position itself advantageously, rather than idly waiting for the next thing to do.

We consider the following model: whenever a request arrives, the server algorithm must service it and the charge is the standard notion of cost. However, once the request is serviced, the server algorithm may adjust its configuration as desired without charge (we also consider the situation in which the free time is bounded). The cost of running a server algorithm with free time is compared with the cost of running the optimal off-line server algorithm *without* free time.[3] At first glance this comparison might seem unfair, but in fact we show that for deterministic algorithms free time helps by at most a small constant factor.

We give an $O(\log^2 k)$-competitive algorithm for general $(k+1)$-point spaces in the free-time model by reducing the problem to a standard server problem on a weighted cache space. We also show that the $\Omega(\log k)$ lower bound for arbitrary weighted cache spaces mentioned above generalizes to algorithms with free time.

[3]For server problems, the optimal off-line cost *with* free time is 0.

Unlike the standard on-line model for which there exists a general $\Omega(\sqrt{\log k/\log\log k})$ lower bound [BKRS92], we show that there exist metric spaces in which one can achieve a constant competitive ratio in the free-time model. (Interestingly, these are exactly the types of spaces proven to have an $\Omega(\log k)$ lower bound in the standard model by [BKRS92].)

In addition, we show that even with free time, there is an $\Omega(k)$ lower bound on the competitive ratio for any deterministic algorithm and any space, and thus, without randomization, free time helps by at most a constant factor.

In the free-time model an algorithm may prepare in any way it likes for future requests although it has no knowledge about what those future requests might be. A natural variant of this model gives the server algorithm some access to information about future requests in the form of possibly erroneous *hints*. Some care is needed to make a meaningful definition of hints. We describe a natural model in which there is a free-time server algorithm for general spaces with an $O(\log k + (1-p)\log^2 k)$ competitive ratio if the hints are correct with probability $p$.

It is also reasonable to assume that in many cases the free time available to an on-line algorithm is bounded. That is, only a limited amount of free work can be done between requests. We show that, even so, in some circumstances, bounded free time provides all the benefits of unlimited free time. In particular, for the $k$-server problem on $(k+1)$-point metric spaces corresponding to unweighted graphs, free time that is only logarithmic in the diameter of the space is sufficient to provide all the benefits of unlimited free time (up to constant factors). Furthermore, even if free time is limited to be constant, the competitive ratio increases at most logarithmically with the diameter.

Finally, we present an algorithm that a computer might use to pre-process potential future commands while waiting for a user to type. The algorithm takes into account the relative durations of possible future instructions.

## 7.6 Related Work

The general $k$-server problem was first presented by Manasse, McGeoch and Sleator [MMS88b], who also show a lower bound of $k$ on the competitive ratio of any deterministic server algorithm.

The uniform space, described above in Section 7.3.1, was initially studied by Sleator and Tarjan [ST85], who showed that LRU and FIFO are $k$-competitive. Randomized approaches for the uniform space began with Fiat *et al.*'s *marking algorithm* [FKM$^+$91], a $2H_k$-competitive algorithm where $H_k \cong \ln k$ is the $k^{th}$ harmonic number. They also show a $\log k$ lower bound on the competitive ratio of any randomized algorithm for the uniform space. McGeoch and Sleator later improved the upper bound to $H_k$, matching the lower bound.

There are several results known for weighted caching, described in Section 7.4, a natural extension of the standard caching problem that corresponds to the $k$-server problem on the uniform space. The problem was first described by Manasse, McGeoch and Sleator [MMS88a] as an example of an asymmetric server problem. In [RS89] Raghavan and Snir presented their *harmonic algorithm*, an $O(k)$-competitive randomized algorithm and the first competitive algorithm for the problem. Subsequently Chrobak, Karloff, Payne and Vishwanathan [CKPV90] gave an $O(k)$-competitive deterministic algorithm called the BALANCE algorithm, and presented some hardness results involving asymmetric metric spaces. Young [You91] studied the problem in the context of approximative primal-dual algorithms, and generalized the result to adversaries with fewer than $k$ servers. Several of these authors ([MMS88b, You91]) suggested that randomized algorithms with substantially better than linear competitive ratios might be possible; however, no algorithms or lower bounds were known for randomized weighted caching (even in the case of $(k+1)$ points) that were any better than the bounds for general spaces. We give an $O(\log^2 k)$-competitive algorithm for $(k+1)$-point spaces.

There are a number of other metric spaces for which deterministic algorithms have been studied. Chrobak and Larmore [CL91] present a $k$-competitive deterministic algorithm for any metric space that is a tree, meaning a planar embedding of a free tree. The distance between two points is simply the arc length of the unique path connecting the points in the tree. Chrobak, Karloff, Payne and Vishwanathan [CKPV90] give a $k$-competitive deterministic algorithm called *double coverage* for scheduling on the line, as described in Section 7.3.1. Karlin, Manasse, McGeoch and Owicki [KMMO94] consider the spin-block problem described in Section 7.3.1. They give a randomized algorithm for the problem with competitive ratio approaching $e/(e-1) \cong 1.58$, an improvement upon the traditional deterministic algorithm which has (optimal deterministic) competitive ratio 2.

In the deterministic setting, Papadimitriou and Koutsoupias [PK94] proved a long-standing conjecture that the *work-function algorithm* has competitive ratio polynomial in $k$ for every metric space. In fact, they showed a competitive ratio of $2k-1$, within a factor of 2 of optimal for every space.

At this point, it became interesting to know whether randomized algorithms like the marking algorithm for the uniform space could be developed to give sub-linear competitive ratios for other spaces, especially in the task system domain as a first step towards the general $k$-server problem.

Two types of lower bounds are known for randomized MTS algorithms. For certain specific metric spaces such as the uniform space studied by Borodin, Linial and Saks [BLS92] and the super-increasing space of Karloff, Rabani and Ravid [KRR91] there are

$\Omega(\log k)$ lower bounds on the competitive ratio of any online algorithm. We show an $\Omega(\log k)$ lower bound for any weighted cache space. A weaker bound of $\Omega(\log \log k)$ due to Karloff, Rabani and Ravid [KRR91], subsequently improved to $\Omega\left(\sqrt{\log k / \log \log k}\right)$ by Blum, Karloff, Rabani and Saks [BKRS92], applies to every metric space.

Irani and Seiden give an $en/(e-1)$-competitive randomized MTS algorithm [IS95] for any metric space. Early results on specific spaces have focused largely on the uniform space: Borodin, Linial and Saks [BLS92] give a $2H_n$ upper bound and an $H_n$ lower bound. Irani and Seiden [IS95] give an $H_n + O(\sqrt{\log n})$-competitive algorithm, matching the lower bound to within lower-order terms. We give an $O(\log^2 n)$-competitive algorithm for any weighted-cache task system. Finally, later work by Blum, Bartal, Burch and myself [BBBT96] gives an $O(\log^6 n)$-competitive algorithm for any metric space.

Fiat and Ricklin [FR94] study a very different problem with a similar flavor, the *weighted-server problem.* In their model the weights apply to the servers rather than to the nodes of the graph, and the cost of moving a server across a distance is scaled by the server's weight. This is a substantially harder problem than the traditional $k$-server problem; in fact, Fiat and Ricklin show that for any metric space, even if the servers are limited to only two possible weights, there is an assignment of weights to servers such that the competitive ratio of any algorithm is at least exponential in $k$.

# Chapter 8

# Weighted Caching

*From earliest times such competitive games had been celebrated.*

*— L. M. Mitchell*

The weighted caching problem is presented in detail, with known results, in Sections 7.4 and 7.6. I recap the necessary definitions here. The *weighted caching problem* is a $k$-server problem in which the cost to move a server to a point is equal to the *weight* of the point, regardless of the source: $d_{ij} = w_j$.

This chapter gives an $O(\log^2 k)$-competitive randomized algorithm for weighted caching on $(k+1)$-point spaces (also known Cat-and-Mouse problems or Pursuit-Evasion games [BKRS92]). We also give an $\Omega(\log k)$ lower bound on the competitive ratio of any such algorithm, extending and simplifying results of [KRR91].

All work described in this chapter was joint with Avrim Blum and Merrick Furst. The techniques described here, combined with a recent result of Yair Bartal's [Bar96], led (with substantial modifications and improvements) to a much more general result, solving the problem described here not just for weighted cache spaces, but for any metric space (with an increase from $O(\log^2 k)$ to $O(\log^6 k)$ in the competitive ratio). That work was joint between Yair Bartal, Avrim Blum, and Carl Burch, and myself, and is described in [BBBT96]. The work described here benefited from the presentation worked out in [BBBT96], and in particular from Carl Burch's suggestion of a potential function for weighted cache spaces.

## 8.1 Definitions and Preliminaries

An algorithm for the $k$-server problem on a $(k+1)$-point metric space can be viewed as controlling the position of a *hole* (the point without a server) which moves whenever its location is hit by a request.

We consider randomized algorithms in the oblivious adversary model. In this model, the adversary does not know the location of the hole, but it does know the online algorithm. Therefore at each point in time, the adversary knows the *distribution* of the location of the hole, as induced by the algorithm, but does not know which element of the distribution was chosen by the algorithm's coins. It will be convenient to view a randomized algorithm as maintaining a set of probability masses, one for each point in the space, that sum to 1 and correspond to the location of the hole. If the algorithm wishes to move a probability mass $p$ for a distance $d$ (for instance, when the point at which the probability mass is located is requested), then it pays a cost $pd$ to do so.

Given a sequence of requests, define $\mathrm{OPT}(i)$ to be the optimal off-line cost of servicing the requests and ending with the hole at point $i$.[1] Notice that it is always the case that $|\mathrm{OPT}(i) - \mathrm{OPT}(j)| \leq d_{ij}$, and that a request to point $i$ does not change $\mathrm{OPT}(j)$ for $j \neq i$.

## 8.2 The Super-Increasing Algorithm

This section gives an $O(\log k)$-competitive randomized algorithm for a particular $(k+1)$-point space called the *super-increasing space*. Section 8.3 uses this algorithm to develop an $O(\log^2 k)$-competitive randomized algorithm for any $(k+1)$-point weighted cache space.

### 8.2.1 Overview of the Super-Increasing Algorithm

The $(k+1)$-point super-increasing space is defined as follows.[2] The points are labeled $0, 1, \ldots k$ and for $j < i$, the distance $d_{ij} = d_i = 2^{i-2}$. Point 0 is called the origin. For this space I show the following theorem:

**Theorem 5** *There is an $O(\log k)$-competitive algorithm for the super-increasing space on $(k+1)$ points.*

The high-level idea of the algorithm is to assume recursively that one has an $r_{i-1}$-competitive algorithm for the super-increasing space on points $0, 1, \ldots, i-1$, and to use that to create an $r_i = (r_{i-1} + c_1/c_2^{r_{i-1}})$-competitive algorithm when point $i$ is added, for some constants $c_1, c_2 > 1$. This will suffice to show the theorem, as the following intuition shows. Consider the smallest positive $i_0$ such that $r_{i_0} \geq \log_{c_2} k$. For all $i \geq i_0$

[1] This value is sometimes called the *work function*.

[2] This definition is somewhat different from the super-increasing space used by [KRR91] in which distances increase at an even faster rate.

we must have $r_i \le r_{i+1} \le r_i + O(1/k)$. Therefore, the final ratio, $r_k$, can be no larger than $O(\log k)$.

Thus, the difficulty is to add a single $i^{th}$ point to an $(i-1)$-point space with only a tiny increase in the competitive ratio. Recall that the "hole" of a space is the single point of the space not occupied by a server. Consider a sequence of accesses to the initial $(i-1)$-point space, followed by a single access to point $i$. The algorithm begins with its hole in the $(i-1)$-point space. As the adversary continues to request points from that space, the algorithm slowly moves its probability mass (the probability of the hole being located at any given point) towards point $i$, using probabilities based on the algorithm of [KMMO94] for 3-point spaces. Finally, when point $i$ is requested, all the probability mass moves back to the $(i-1)$-point space and the process is repeated.

To counter such an algorithm, the adversary may adopt one of two strategies. First, it may plan to move its hole to point $i$ immediately, and the sequence will contain a large number of requests to the inner space that will be free for the offline solution. Second, it may plan never to move its hole to point $i$, and may present only a short sequence of accesses to the inner space followed by a request for point $i$. If the algorithm moves its probability mass too slowly to point $i$ then the adversary will adopt the first strategy; if the algorithm moves its probability mass too quickly, the adversary will adopt the second strategy. The algorithm must balance these two costs so the adversary will have no obvious winning strategy.

This is the same high-level idea used in [BKRS92] for "sufficiently unbalanced" spaces. One difficulty in this case, however, is that because distances are growing only by factors of 2 (as opposed to factors of $p(i)$ for a sufficiently large polynomial $p$ as required by [BKRS92]) we need to be particularly careful about our inductive assumptions and the way in which the algorithm for an $i$-point space interacts with the algorithm for a larger containing space.[3]

Let $\mathrm{Alg}_i$ denote the algorithm for the space of points $\{0, 1, \ldots, i\}$. $\mathrm{Alg}_i$ will be given a probability mass (that may be less than 1) which it is responsible for distributing across the points in its space. $\mathrm{Alg}_i$ will also need to handle requests by $\mathrm{Alg}_{i+1}$ to either give or receive probability mass.

In fact, the algorithm for the space $\{0, 1, \ldots, i\}$ will be explicitly aware that its total probability mass is some quantity $P$ that may be less than 1. This value $P$ in general will be slowly decreasing as probability gets drawn out of the space by the parent algorithms. Unfortunately, $\mathrm{Alg}_{i+1}$, $\mathrm{Alg}_{i+2}$ and so on may also pass probability back to $\mathrm{Alg}_i$. We will

[3] One difficulty roughly is as follows. We would like to assume inductively that for each increment in the off-line cost, the algorithm for points $0, 1, \ldots, i-1$ pays only $r_{i-1}$. However, the result for the $(i+1)$-point space will be just an algorithm whose *amortized* cost per off-line increment is $r_{i-1} + c_1/c_2^{r_{i-1}}$. Therefore we need to put some sort of amortization into our assumption, and be careful to maintain it inductively.

adopt the invariant that all such probability is placed at point $i$, and a potential function always contains enough potential to move all probability at point $i$ back to $\mathrm{Alg}_{i-1}$ without charge.

In order to show competitiveness, I require a formal description of the offline cost. At any point in time, the offline cost is exactly the minimum of the OPT values. But the OPT values never differ from one another by more than the diameter of the space, which is a fixed constant independent of the sequence. So it is common to adopt a particular OPT value, or even a convex combination of the OPT values, as the "official" definition of offline cost against which the algorithm must compete. In this case, we take OPT(0) to be the offline cost. Whenever point 0 is requested, $\mathrm{Alg}_i$ will incur a *local cost* for rearranging probability within $\mathrm{Alg}_{i-1}$ and a *movement cost* for moving probability to point $i$. The movement cost will also contribute to a potential function with the property that there is always enough potential to move probability *back* from point $i$ to $\mathrm{Alg}_{i-1}$ without cost. Thus, whenever point 0 is requested, I will show that the offline cost increases by 1 and the online cost increases by no more than $r_i$. When point $i$ is requested, the offline cost will not increase (since OPT(0) will not increase), but the amortized cost to the algorithm will be 0 since the change in potential will entirely cover the cost of moving probability from point $i$ to $\mathrm{Alg}_{i-1}$.

### 8.2.2 Formal Description of the Super-Increasing Algorithm

Define $r_1 = 1$ and $r_i = c \log i$ for $i > 1$, for some sufficiently large constant $c$.

$\mathrm{Alg}_i$ for $i \geq 2$ is defined as follows. Let $r = r_{i-1}$ and $d = d_i$. Initially, and any time point $i$ is requested, all of its probability mass is given to $\mathrm{Alg}_{i-1}$. By definition of the super-increasing space, the value of OPT(0) increases in unit increments. The first time that OPT(0) is incremented by 1, $\mathrm{Alg}_i$ moves a fraction $\frac{c}{de^{r/4}}$ of its probability mass from $\mathrm{Alg}_{i-1}$ to point $i$, where $c$ is a sufficiently large constant. More generally, the $t^{th}$ time that OPT(0) is incremented by 1, $\mathrm{Alg}_i$ moves the following fraction of its total probability mass from $\mathrm{Alg}_{i-1}$ to point $i$:[4]

$$frac\_moved(i,t) = \frac{c}{de^{r/4}}\left(1 + \frac{r}{3d}\right)^{t-1} \tag{8.1}$$

If $\mathrm{Alg}_i$ is ever asked to give probability mass to $\mathrm{Alg}_{i+1}$, then it does so by reducing all its probabilities at a uniform rate (e.g., if it has $P$ units of probability and is asked for $P/2$ probability mass, then it reduces the probability at point $i$ by half and it asks $\mathrm{Alg}_{i-1}$ for half its probability mass). If $\mathrm{Alg}_i$ is ever given probability from $\mathrm{Alg}_{i+1}$ then it places all the mass at point $i$.

[4] If this fraction is larger than the fraction of probability remaining within $\mathrm{Alg}_{i-1}$ then it simply asks for all of $\mathrm{Alg}_i$'s probability mass.

### 8.2.3 Competitiveness of the Super-Increasing Algorithm

Let us assume without loss of generality that the adversary never requests a point whose probability mass is zero. The following is a useful lemma about our rules for moving probabilities.

**Lemma 1** *If $OPT(i) \leq OPT(0) - d_i/2$ then the super-increasing algorithm places zero probability on any of the points $j < i$.*

**Proof:** Assume inductively that the lemma is true for points $i+1, i+2, \ldots, k$.

First, notice that after point $i$ is requested, $\mathrm{OPT}(i) \geq \mathrm{OPT}(0) + d_i/2$. The reason is that if $i$ is requested, then $\mathrm{OPT}(i) = \min_{j \neq i} \mathrm{OPT}(j) + d_{ij}$. For $j > i$, by our inductive assumption and the assumption that points of probability mass 0 are not hit we have $\mathrm{OPT}(j) + d_{ij} > \mathrm{OPT}(0) - d_j/2 + d_j > \mathrm{OPT}(0) + d_i/2$. For $j < i$ we have $\mathrm{OPT}(j) + d_{ij} \geq \mathrm{OPT}(0) - d_j + d_i \geq \mathrm{OPT}(0) + d_i/2$ by definition of the $d_i$'s.

Now, by our rules for moving probability mass, we have that if $\mathrm{OPT}(i) \leq \mathrm{OPT}(0) - d_i/2$ then the total fraction of the probability mass initially given to $\mathrm{Alg}_{i-1}$ that has been moved over to point $i$ is at least:

$$\begin{aligned}
\sum_{s=0}^{d_i-1} \left[ \frac{c}{d_i e^{r/4}} \left(1 + \frac{r}{3d_i}\right)^s \right] &= \frac{c}{d_i e^{r/4}} \left[ \frac{\left(1 + \frac{r}{3d_i}\right)^{d_i} - 1}{\left(1 + \frac{r}{3d_i}\right) - 1} \right] \\
&= \frac{3c}{re^{r/4}} \left[ \left(1 + \frac{r}{3d_i}\right)^{d_i} - 1 \right] \\
&\geq 1. \qquad \text{(for sufficiently large } c\text{)}
\end{aligned}$$

■

The above lemma is needed for two reasons. First, it is important that if the values of $\mathrm{OPT}(j)$ for $j < i$ are constrained by $\mathrm{OPT}(i)$, meaning that the adversary can hit them with impunity without increasing the $\mathrm{OPT}(j)$ values, then the algorithm should not have its hole there. Second, it is important that for any $j \geq 0$, for all $i \geq j$, all the algorithms $\mathrm{Alg}_i$ agree on the value of $\mathrm{OPT}(j)$ in order for the induction to go through. This is formalized in the following lemma.

**Lemma 2** *During the course of the algorithm, for all $0 \leq j < i$, $OPT(j) < OPT(i) + d_i$. That is, $OPT(j)$ is never constrained by $OPT(i)$ for $i > j$.*

**Proof:** Using our assumption that the adversary never requests a point that has zero probability, if point $j$ is requested then by Lemma 1 we have: $\mathrm{OPT}(i) > \mathrm{OPT}(0) - d_i/2 \geq \mathrm{OPT}(j) - d_j - d_i/2 \geq \mathrm{OPT}(j) - d_i$. ■

With these two lemmas in place, I can restate and prove the main theorem of this section.

**Theorem 5** *There is an* $O(\log k)$*-competitive algorithm for the super-increasing space on* $(k+1)$ *points.*

**Proof:**

As described above, a potential function will amortize the cost of moving probability from point $i$ to $\mathrm{Alg}_{i-1}$. Let $p_i$ be the probability mass at point $i$. The potential function for $\mathrm{Alg}_i$, $i > 1$, is defined as

$$\Phi_i = \Phi_{i-1} + 2p_i d_i.$$

The potential $\Phi_0$ of a 1-point space is 0, and the potential $\Phi_1$ of a 2-point space is $p_1 d_1$. $\Phi_k$, the potential for the entire space, is given by

$$\Phi_k = p_1 d_1 + \sum_{i=2}^{k} 2p_i d_i \leq 2d_k.$$

Define $\mathrm{C}_{\mathrm{amort}}(\mathrm{Alg}_i)$ to be the *amortized cost to* $Alg_i$, or the actual cost incurred by moving probability both within $\mathrm{Alg}_{i-1}$ and to or from point $i$, plus any change in potential. Let $\Pr[\mathrm{Alg}_i]$ be the probability mass within $\mathrm{Alg}_i$. We will inductively prove the following property:

**Property 1** *Whenever any point* $j \neq 0$ *is requested, the amortized cost to* $Alg_i$ *is 0. Whenever point 0 is requested, so* $OPT(0)$ *increases by 1, the amortized cost to* $Alg_i$ *is no more than* $\Pr[Alg_i] \cdot r_i$.

For the base case, $\mathrm{Alg}_1$ simply moves all its probability mass back and forth between points 0 and 1. If the algorithm has $p$ probability at point 0, and point 0 is requested, the algorithm must pay $pd_1 = p/2$. The change in potential is $pd_1 = p/2$, so the total cost to $\mathrm{Alg}_1$ is $p$. The offline cost increases by exactly 1 whenever OPT(0) is requested, and so we can take $r_1 = 1$, which is constant and therefore acceptable.

For the recursive case, I begin with a few definitions. The *current phase* of $\mathrm{Alg}_i$ is the set of requests following the most recent request to point $i$ (or the entire sequence so far if point $i$ has not yet been requested). Thus, a phase consists of a sequence of requests to $\mathrm{Alg}_{i-1}$ followed by a single request to point $i$, with possible intervening requests to points $j > i$. Note that, until the phase ends, probability will leave for point $i$ according to Equation 8.1, and will leave for points $j > i$ when enclosing algorithms draw probability out of the entire $i$-point space, but no probability will enter $\mathrm{Alg}_{i-1}$. This is because all probability passed back to $\mathrm{Alg}_i$ from $\mathrm{Alg}_{i+1}$ is deposited at point $i$, and never enters $\mathrm{Alg}_{i-1}$ until the end of $\mathrm{Alg}_i$'s phase. If we ignore any probabiliy that leaves $\mathrm{Alg}_i$, we can write the total fraction of probability that has left $\mathrm{Alg}_{i-1}$ for point $i$ after the $t^{th}$ request of the phase:

$$\begin{aligned}
\mathit{total_frac_moved}(i,t) &= \sum_{j=1}^{t} \mathit{frac_moved}(i,j) \\
&= \sum_{j=1}^{t} \frac{c}{de^{r/4}} \left(1+\frac{r}{3d}\right)^{j-1} \\
&= \frac{3c}{re^{r/4}} \left[\left(1+\frac{r}{3d}\right)^{t} - 1\right]
\end{aligned}$$

Next, we take into account that some probability may have left $\mathrm{Alg}_i$ altogether, but that no probability from outside $\mathrm{Alg}_i$ may have entered $\mathrm{Alg}_{i-1}$ during this phase. We use this fact to upper bound the actual probability (rather than the *fraction* of probability) moving from $\mathrm{Alg}_{i-1}$ to point $i$ during the $t^{th}$ request of the phase. Let $\Pr[\mathrm{Alg}_{i-1}, t]$ be the probability mass of $\mathrm{Alg}_{i-1}$ after servicing the $t^{th}$ request of the phase, and $\Delta \Pr[\mathrm{Alg}_{i-1}, t]$ be the amount of probability that moves from $\mathrm{Alg}_{i-1}$ to point $i$ during the $t^{th}$ request of the phase. Then:

$$\Delta \Pr[\mathrm{Alg}_{i-1}, t] \;\leq\; \Pr[\mathrm{Alg}_i, t]\mathit{frac\_moved}(i,t) \tag{8.2}$$

Next, we bound $\Pr[\mathrm{Alg}_{i-1}, t]$. Since a *fraction* of the current probability mass of $\mathrm{Alg}_i$ departs $\mathrm{Alg}_{i-1}$ at each round, the current split of probability between $\mathrm{Alg}_{i-1}$ and point $i$ is not influenced by mass drawn out from $\mathrm{Alg}_i$. This is because mass is drawn out from each piece according to the mass of that piece. If no probability has been deposited at point $i$ by $\mathrm{Alg}_{i+1}$ then after $t$ requests to point 0, the fraction of $\Pr[\mathrm{Alg}_i, t]$ residing at $\mathrm{Alg}_{i-1}$ will be $\mathit{total\_frac\_moved}(i,t)$. If $\Pr[\mathrm{Alg}_i]$ has increased since the start of the phase by mass arriving at point $i$ from $\mathrm{Alg}_{i+1}$, then during a particular request more probability may leave $\mathrm{Alg}_{i-1}$, and the *actual* probability (rather than the fraction) at $\mathrm{Alg}_{i-1}$ may actually be smaller:

$$\Pr[\mathrm{Alg}_{i-1}, t] \;\leq\; \Pr[\mathrm{Alg}_i, t](1 - \mathit{total\_frac\_moved}(i,t)) \tag{8.3}$$

The amortized cost to $\mathrm{Alg}_i$ in servicing the $t^{th}$ request to point 0 during a phase has three components: the *local cost* within $\mathrm{Alg}_{i-1}$, the *movement cost* of moving probability to point $i$, and the change in $\Phi_i$. I consider each in turn. First, let $\mathrm{C_{actual}}(\mathrm{Alg}_{i-1}, t)$ be the local cost to $\mathrm{Alg}_{i-1}$, the actual (non-amortized) cost incurred by $\mathrm{Alg}_{i-1}$ in servicing a request to point 0. Since the amortized cost by induction is $\mathrm{C_{amort}}(\mathrm{Alg}_{i-1}, t) = \mathrm{C_{actual}}(\mathrm{Alg}_{i-1}, t) + \Delta\Phi_{i-1} \leq r \Pr[\mathrm{Alg}_{i-1}, t]$, we can write the actual cost as $\mathrm{C_{actual}}(\mathrm{Alg}_{i-1}, t) \leq r \Pr[\mathrm{Alg}_{i-1}, t] - \Delta\Phi_{i-1}$. Second, the movement cost will be exactly $d\Delta \Pr[\mathrm{Alg}_{i-1}, t]$. Third, we consider the change to $\Phi_i$: the $2d_i p_i$ term will increase by $2d\Delta \Pr[\mathrm{Alg}_{i-1}, t]$ as more probability arrives at point $i$, and the $\Phi_{i-1}$ term will increase by $\Delta\Phi_{i-1}$.

We can now complete the analysis:

$$\begin{aligned}
\mathrm{C_{amort}}(\mathrm{Alg}_i, t) &= (\text{local cost within } \mathrm{Alg}_{i-1}) + (\text{movement cost to point } i) + (\Delta\Phi_i) \\
&\leq \overbrace{\Pr[\mathrm{Alg}_{i-1}, t]r - \Delta\Phi_{i-1}}^{\text{local cost}} + \overbrace{d\Delta \Pr[\mathrm{Alg}_{i-1}, t]}^{\text{movement cost}} + \\
&\qquad \overbrace{2d\Delta \Pr[\mathrm{Alg}_{i-1}, t] + \Delta\Phi_{i-1}}^{\Delta\Phi_i} \\
&\leq \Pr[\mathrm{Alg}_i, t](1 - \mathit{total_frac_moved}(i,t))r + d\Pr[\mathrm{Alg}_i, t]\mathit{frac_moved}(i,t) + \\
&\qquad 2d\Pr[\mathrm{Alg}_i, t]\mathit{frac_moved}(i,t) \\
&\leq \Pr[\mathrm{Alg}_i, t](1 - \mathit{total_frac_moved}(i,t))r + 3d\Pr[\mathrm{Alg}_i, t]\frac{c}{de^{r/4}}\left(1 + \frac{r}{3d}\right)^{t-1} \\
&\leq \Pr[\mathrm{Alg}_i, t]r - \Pr[\mathrm{Alg}_i, t]r\frac{3c}{re^{r/4}}\left[\left(1 + \frac{r}{3d}\right)^{t} - 1\right] + \frac{3c}{e^{r/4}}\left(1 + \frac{r}{3d}\right)^{t-1} \\
&\leq \Pr[\mathrm{Alg}_i, t]r + \Pr[\mathrm{Alg}_i, t]\frac{3c}{e^{r/4}} \\
&= \Pr[\mathrm{Alg}_i, t]\left(r + \frac{3c}{e^{r/4}}\right) \\
&= \Pr[\mathrm{Alg}_i, t]r_i
\end{aligned}$$

Globally, the derivation above, including the idea expressed in Equation 8.2, assumes that the nested algorithms work as follows. When point 0 is requested, first $\mathrm{Alg}_k$ draws as much probability as it chooses from $\mathrm{Alg}_{k-1}$. The probability mass at points within $\mathrm{Alg}_{k-1}$ decreases, which results in a decrease in $\Phi_{k-1}$, but we allow this decrease to take place without offsetting it against anything. Of course, the global potential increases, because all that probability ends up occupying a more expensive point (a point with a larger $d$ value), but the increase to the potential is paid for by the movement of $\mathrm{Alg}_k$, and does not require a more careful accounting of the internal potentials. Once this operation completes, $\mathrm{Alg}_{k-1}$ rearranges itself by drawing probability from $\mathrm{Alg}_{k-2}$ to point $k-1$, using the movement to pay for an increase in $\Phi_{k-1}$. Thus, the local cost to $\mathrm{Alg}_{i-1}$ is scaled by the resulting probability $\Pr[\mathrm{Alg}_{i-1}, t]$ rather than the initial probability $\Pr[\mathrm{Alg}_{i-1}, t-1]$.

It remains to show that accesses to points other than point 0 have amortized cost 0. Again, let $i > 0$ be some other point. The actual cost incurred by $\mathrm{Alg}_i$ in servicing a hit to point $i$ is $p_i d_i$ to move all the probability to point $i-1$. The change in potential is $\Delta\Phi_i = -2p_i d_i + \Delta\Phi_{i-1}$ where the only change to the potential in $\mathrm{Alg}_{i-1}$ is due to the arrival of $p_i$ units of probability at point $i-1$: $\Delta\Phi_{i-1} = 2p_i d_{i-1}$. The overall cost is therefore:

$$\begin{aligned}
\mathrm{C}_{\mathrm{amort}}(\mathrm{Alg}_i) &= \overbrace{p_i d_i}^{\text{actual cost}} + \overbrace{-2p_i d_i + 2p_i d_{i-1}}^{\Delta\Phi_i} \\
&= p_i d_i - 2p_i d_i + 2p_i(d_i/2) \\
&= 0
\end{aligned}$$

This shows that Property 1 is maintained, and therefore completes the proof of Theorem 5.

■

We now use this algorithm as a black box in developing the general algorithm for $(k+1)$-point metric spaces.

## 8.3 The Mark-And-Jump Algorithm

In this section we present the *Mark-And-Jump Algorithm* for $(k+1)$-point weighted cache spaces. We show a competitive ratio of $O(\log^2 k)$.

Let $M = \{m_1, m_2, \ldots, m_{k+1}\}$ be a weighted cache metric space and let $w_i$ be the weight of the $i^{th}$ point. We adopt the formulation that $d_{ij} = w_i$, so when the hole is at $m_i$ and the adversary requests $m_i$, the algorithm will pay $w_i$ to service the request.

We break $M$ into a number of levels $L_j$ as follows. $L_j = \{m_i | 2^{j-3} \leq w_i < 2^{j-2}\}$. Each level can be treated as a uniform space, since the cost to service a miss at any point is within a factor of two of any other point in the level.

The Mark-And-Jump algorithm will use the algorithm of Section 8.2 for the super-increasing space as a black box. We will call that algorithm $\mathcal{S}$. Intuitively, each point of the super-increasing space used by $\mathcal{S}$will correspond to a level of the actual space $M$. Our algorithm will make requests to $\mathcal{S}$whenever certain conditions are met in $M$, and will then move the hole between levels whenever $\mathcal{S}$moves its hole between points.

Our algorithm will also use the Marking Algorithm of [FKM+91]; for completeness I define the algorithm here. The $k$ servers begin on *marked* points $1 \ldots k$. Whenever a point is requested, it is marked. Whenever $k+1$ points are marked, all marks except the most recent one are erased. If the requested point has no server then a server is chosen uniformly from the servers occupying unmarked vertices.

The Mark-And-Jump algorithm begins with the hole at level 1. It schedules on each level using the Marking Algorithm, although of course it does not pay at levels that do not contain the hole. Once an entire level (say $L_j$) is marked, the algorithm issues a request for point $j$ to $\mathcal{S}$. We maintain the invariant that if $\mathcal{S}$'s hole is at point $i$ then the hole of the Mark-And-Jump algorithm is in level $L_i$. If $\mathcal{S}$moves its hole, the Mark-And-Jump algorithm will do likewise.

More formally, all points of $M$ begin unmarked. When a request arrives for a vertex in $L_j$, we first mark the vertex. If some point of the level is not yet marked (*i.e.*, this request did not mark the entire level) then if the vertex has a server we return, and otherwise we move a server uniformly from those servers located at unmarked vertices of $L_j$, and then return.

If all points of $L_j$ are now marked, we remove all but the most recent of those marks. We then submit a request for point $j$ to $\mathcal{S}$ and if the hole is at $L_j$ we move the hole to a uniformly chosen unmarked vertex of the level chosen by $\mathcal{S}$. This guarantees that the requested vertex has a server. We show the following theorem:

**Theorem 6** *The Mark-And-Jump algorithm is* $O(\log^2 k)$*-competitive.*

**Proof:**

Since the number of points at each level is bounded by $k$, the Marking Algorithm is $O(\log k)$-competitive against any adversary that keeps its hole at that level. Intuitively, at each level $L_i$ we incur a $\log |L_i|$-factor loss, and our total cost at each level incurs a log(number of levels)-factor loss from the algorithm for the super-increasing space.

Let $\tau = \tau_1\tau_2 \ldots$ be a sequence of requests in $M$ being serviced by the Mark-And-Jump algorithm. Since $\tau$ is fixed, there is a fixed sequence $\sigma = \sigma_1\sigma_2 \ldots$ of requests to $\mathcal{S}$, where, for instance, $\sigma_1$ is a request for point $j$ such that level $L_j$ is marked before any other level.

First we relate the cost to the optimal off-line algorithms for $M$ under $\tau$, and for the super-increasing space under $\sigma$:

**Lemma 3** $OPT(\sigma) \leq 4OPT(\tau)$

**Proof:** Given an off-line strategy for sequence $\tau$ we will find an off-line strategy for sequence $\sigma$ whose cost is at most 4 times larger. For simplicity, we may assume that the off-line strategy for $\tau$ is lazy; that is, it moves a server only when its hole is requested. If the algorithm servicing $\tau$ begins at a point in level $L_i$, we begin at point $i$ of the super-increasing space. We assume further that all distances within level $L_i$ are exactly $d_i = 2^{i-2}$; this will change the optimal off-line cost by at most a factor of 2. Thus, we may assume that the off-line strategy for $\tau$ always places the hole at the point of a level that will be requested last. Once its hole is hit, all elements of the level are marked so a request is generated in $\sigma$. If the algorithm for $\tau$ moves its hole to level $L_j$, we move our hole to point $j$. If $j = i$ we pay twice as much as the strategy for $\tau$ because we must move away and return; otherwise we pay the same amount. Continuing in this way, the

off-line cost for $\sigma$ is no more than twice the cost for servicing $\tau$ with cost $d_i$ at each level, which is no more than twice the cost for servicing $\tau$ as given; thus, $\mathrm{OPT}(\sigma) \leq 4\mathrm{OPT}(\tau)$.
■

Next, we relate the online costs of $\mathcal{S}$and the Mark-And-Jump algorithm. Let MARK-AND-JUMP be the Mark-And-Jump algorithm for $M$.

**Lemma 4** MARK-AND-JUMP$(\tau) \leq (H_k + 1)\mathcal{S}(\sigma) + c$

**Proof:**

The super-increasing algorithm as given will move probability from $\mathrm{Alg}_{i-1}$ to point $i$ when point 0 is hit. This means that a server could move from point $k-1$ to point $k$ as a result of a request to point 0. We assume a lazy version of the algorithm that remembers the actual location of the hole in the standard version of the algorithm, but waits to actually move the server until the hole is hit. Clearly, the cost incurred by the lazy version will be no greater than the cost incurred by the original. In the lazy version, we observe that if the hole of MARK-AND-JUMP is in level $L_i$ for part of a phase of $L_i$ then it must be in $L_i$ for the entire phase, as the hole moves between levels only as a result of requests to $\mathcal{S}$, which occur only at the end of a phase. Since the algorithm is lazy, if the hole is in $L_i$ and a phase of the Marking Algorithm ends for $L_j$, the hole will not move from $L_i$ as a result of the request generated by $L_j$.

Next we note that if the hole is in $L_i$ then the corresponding hole of $\mathcal{S}$must be at point $i$, so when the phase of $L_i$ completes, $\mathcal{S}$will pay at least $d_i$. The analysis of the Marking Algorithm from [FKM+91] makes it clear that the expected cost to the algorithm over the course of a phase is bounded by the cost to move $H_k$ servers, with no amortization (the proof of [FKM+91] uses amortization in the lower bound on the cost to the offline algorithm, but we require only a bound on the online algorithm's cost here). Each phase of the marking algorithm that actually incurs cost (because the hole is at that level during the phase) can therefore be associated with the request to $\mathcal{S}$at the end of the phase, and the cost incurred by the Marking Algorithm is bounded by $H_k$ times the cost incurred by $\mathcal{S}$to service the request. If the final phase of the sequence does not complete, there will be no request to $\mathcal{S}$with which it is associated, which introduces a constant additional additive cost of as much as $c = H_k d_{\max}$, independent of $\sigma$. Therefore, the cost incurred by moving servers within a level is bounded from above by $H_k\mathcal{S}(\sigma) + c$. Likewise, the cost incurred to move servers between levels is no more than the cost incurred by $\mathcal{S}$, since each point within a level has weight no greater than the weight of the associated point of $\mathcal{S}$. Thus, the total cost to MARK-AND-JUMP is no greater than $H_k\mathcal{S}(\sigma) + c$ for costs within a level, plus $\mathcal{S}(\sigma)$ for costs between levels, for a total of $(H_k + 1)\mathcal{S}(\sigma) + c$.

■

To complete the proof of the theorem, Theorem 5 shows that $\mathcal{S}(\sigma) \leq O(\log k)\mathrm{OPT}(\sigma)$. Therefore, we may string together the two previous lemmas as follows:

$$\begin{aligned}\textsc{Mark-And-Jump}(\tau) &\leq (\log k + 1)\mathcal{S}(\sigma) + c \\ &\leq (\log k + 1)O(\log k)\mathrm{OPT}(seq) + c + c_1 \\ &\leq O(\log^2 k)\mathrm{OPT}(\tau).\end{aligned}$$

■

**Corollary 7** *There is an $O(\log^2 n)$-competitive algorithm for the MTS problem on any weighted-cache space.*

## 8.4 Lower Bounds for Weighted Caching

**Theorem 8** *The competitive ratio of the $k$-server problem on any $(k+1)$-point weighted-cache space is $\Omega(\log k)$.*

**Proof:** Let us first assume that the metric space satisfies the following conditions. The points are labeled $0, 1, \ldots, k$, and the distance is $d_{ij} = \max\{w_i, w_j\}$. Clearly, distances in this space are within a factor of 2 of distances in the traditional version given by $d_{ij} = 1/2(w_i + w_j)$.

Furthermore, we assume that each $w_i$ is a power of 2 and that the $w_i$'s are non-decreasing. Once again, this changes the distances by no more than a factor of 2. We define $w_0$ to equal $w_1$, which does not change the distance at all.

The adversary's (randomized) strategy is simply at each time step to request point $i$ with probability $1/(\alpha w_i)$ where $\alpha = \sum_{i=0}^{k} 1/w_i$. Notice that the expected cost per request of any on-line algorithm is at least $1/\alpha$: if the algorithm is currently at point $i$, then the probability it is hit is $1/(\alpha w_i)$ and if hit it must pay at least $w_i$ to move. Thus, to prove the lower bound we just need to describe an off-line strategy whose average cost per request is $O(1/(\alpha \log k))$.

Consider a long sequence of requests. Partition the sequence into intervals of length $\frac{1}{2}\alpha w_k \ln k$. For each interval, the probability that there is no request to point $k$ inside that interval is $(1 - 1/(\alpha w_k))^{\frac{1}{2}\alpha w_k \ln k} \approx e^{-\frac{1}{2}\ln k} = 1/\sqrt{k}$.

Let us "shade in" those good intervals to represent that we know what to do then: namely, the off-line algorithm will move to point $k$ at the start of the interval, wait out the $\frac{1}{2}\alpha w_k \ln k$ requests, and then move back to the origin, paying cost only $O(w_k)$. Now consider the unshaded intervals. If $w_k = 2^t w_{k-1}$ (where $t$ may equal 0) we split each unshaded interval into $2^t$ equal parts, and we think of each part as an interval of length $\frac{1}{2}\alpha w_{k-1} \ln k$. For each of these intervals, consider the good event $G$ that it contains no request to point $k-1$. Notice that this is the same random process as before, except that we are conditioning on the event that there *was* at least one request to point $k$ inside the original interval. However, this only *increases* the probability of our desired event $G$. (More generally, the probability there is no request to point $i$ in some interval $I$ is only increased if we condition on the event that there *were* requests to points $i_1, i_2, \ldots$ in intervals $I_1, I_2, \ldots \supseteq I$.) Therefore, again there is at least a $1/\sqrt{k}$ chance of each interval being "good" for point $k-1$. As before, we shade in the good intervals, split the unshaded intervals as necessary, and recurse. After we complete this entire process, the expected fraction of the entire request sequence that remains unshaded is $(1-1/\sqrt{k})^k = \Theta(e^{-\sqrt{k}})$. Our final algorithm is in shaded intervals to move to the associated point and pay only $O(1/(\alpha \log k))$ on average per request, and for the remaining requests to simply shuttle between the origin and its nearest neighbor, paying $O(w_1)$ per request. We note that since $\sum_{i=0}^{k} 1/w_i = \alpha$, we must have $w_1 \leq (k+1)/\alpha$, so we are paying at most a factor of $k$ more than the on-line algorithm on the unshaded regions. Because only a small fraction of points are unshaded, our overall average cost per request remains $O(1/(\alpha \log k))$, which is a factor of $\Omega(\log k)$ less than any on-line algorithm.

■

**Corollary 9** *An on-line algorithm for any weighted caching task system has competitive ratio* $\Omega(\log n)$.

# Chapter 9

# Free Time

*Idle hands do the devil's work*

— *Traditional*

In this chapter we describe the free-time model and show its connection to weighted caching. The results in this chapter are all joint with Avrim Blum and Merrick Furst.

## 9.1 Introduction to Free Time

In the traditional model of on-line computation, an algorithm is asked to process a sequence of *requests*. Each request is presented after the algorithm has completed processing the previous one and the cost of the algorithm is the cumulative time or work needed. In many natural on-line settings, however, requests may arrive infrequently in relation to the speed of the algorithm. In these situations, it makes sense to model an algorithm as having free time, for which it is not charged, between the servicing of one request and the arrival of the next. For instance, in the typical "servers are fire trucks" example, one rightly cares much more about the time it takes to get a fire truck to a fire once a call has been made and cares much less about any time spent moving trucks to resting places while there are no fires to attend to. In general, in computing situations, if the process issuing requests is substantially slower than the process serving requests then the server is liable to have a fair amount of free time at its disposal between demands. In these situations it makes sense for the serving algorithm to use the free time between requests to position itself advantageously, rather than idly waiting for the next thing to do.

We consider the following model: whenever a request arrives, the server algorithm must service it and the charge is the standard notion of cost. However, once the request is serviced, the server algorithm may adjust its configuration as desired without charge (in Section 9.6 we consider the situation in which the free time is bounded). The cost of running a server algorithm with free time is compared with the cost of running the optimal off-line server algorithm *without* free time.[1]

In the last section we gave an $O(\log^2 k)$-competitive randomized algorithm for $(k+1)$-point weighted-cache spaces. We will now present an $O(\log^2 k)$-competitive algorithm for general $(k+1)$-point spaces in the free-time model by reducing the problem to a standard server problem on a weighted cache space. We also show that the $\Omega(\log k)$ lower bound for arbitrary weighted cache spaces mentioned above generalizes to algorithms with free time.

Unlike the standard on-line model for which there exists a general $\Omega(\sqrt{\log k / \log\log k})$ lower bound [BKRS92], we show that there exist metric spaces in which one can achieve a constant competitive ratio in the free-time model. (Interestingly, these are exactly the types of spaces proven to have an $\Omega(\log k)$ lower bound in the standard model by [BKRS92].)

In addition, we show that even with free time, there is an $\Omega(k)$ lower bound on the competitive ratio for any deterministic algorithm and any space, and thus, without randomization, free time helps by at most a constant factor.

In the free-time model an algorithm may prepare in any way it likes for future requests although it has no knowledge about what those future requests might be. A natural variant of this model gives the server algorithm some access to information about future requests in the form of possibly erroneous *hints*. Some care is needed to make a meaningful definition of hints. We describe a natural model in which there is a free-time server algorithm for general spaces with an $O(\log k + (1-p)\log^2 k)$ competitive ratio if the hints are correct with probability $p$.

It is also reasonable to assume that in many cases the free time available to an on-line algorithm is bounded. That is, only a limited amount of free work can be done between requests. We show that, even so, in some circumstances, bounded free time provides all the benefits of unlimited free time. In particular, for the $k$-server problem on $(k+1)$-point metric spaces corresponding to unweighted graphs, free time that is only logarithmic in the diameter of the space is sufficient to provide all the benefits of unlimited free time (up to constant factors). Furthermore, even if free time is limited to be constant, the competitive ratio increases at most logarithmically with the diameter.

In the final sections of the paper we present an algorithm that a computer might use to preprocess potential future commands while waiting for a user to type. The algorithm takes into account the relative durations of possible future instructions.

[1]For server problems, the optimal off-line cost *with* free time is 0.

## 9.2 Free Time and Weighted Caching

The *k-server problem with free time* is defined as follows. An algorithm for this problem is presented with elements from a sequence of requests just as in the traditional model. But after a request has been serviced, the algorithm may perform any set of server movements without cost, before the next request is presented.

In this section, we begin by giving a straightforward reduction from general $(k+1)$-point metric spaces with free time to weighted caching without free time. We then explore the model by presenting a series of extensions to other domains, answering some questions, and phrasing some open problems.

**Theorem 10** *For any metric space on $(k+1)$ points with free time, Mark-And-Jump yields an $O(\log^2 k)$-competitive algorithm. For any approximate star space on $(k+1)$ points with free time, no algorithm can have competitive ratio better than $\Omega(\log k)$.*

**Proof:** First we show the upper bound. For any metric space, we associate with each point $j$ a value $w_j = \min_i d_{ij}$. Since we have only $(k+1)$ points, there is only one hole. Whenever the adversary requests a point $j$ that is the location of the hole, the algorithm will move the server from the nearest point $i$ (at cost $w_j$ — we adopt the variant of weighted caching in which $d_{ij} = w_j$) and will then move the hole to a more desirable location during free time.[2] Thus, any algorithm for the induced weighted cache space can be applied with no additional cost to the original space with free time. We must also show that the optimal off-line cost for a request sequence in the weighted cache space is no greater than the cost to the off-line adversary in the original space. But this follows immediately because distances in the original space are at least as great as distances in the weighted cache space, and the off-line adversary is not allowed to use free time.

Second we show the lower bound. The proof of Section 8.4 presents a particular sequence. The off-line cost in the free time model is identical to the off-line cost in the original model. So we must show that the calculation of expected cost to the algorithm (over the randomly chosen sequences) is still valid when the algorithm is augmented with free time. This is clear because after the free time finishes, there will be some distribution over the points of the space describing the position of the hole, and the proof of the lower bound holds for any such distribution (we are using the fact that, during free time, the algorithm has not seen the next request.) This completes the theorem. ∎

[2] Note that this reduction does not require symmetry in the original space.

## 9.3 Deterministic Algorithms With Free Time

In this section we show that deterministic algorithms with free time cannot perform substantially better than their counterparts in the traditional model. Specifically, we show that for any metric space on at least $(k+1)$ points, no deterministic server algorithm with free time can be better than $(k+1)/2$-competitive. In light of results of [PK94], who show that the Work Function algorithm for the standard $k$-server problem is $2k-1$-competitive, free time provides at best a small improvement.

**Theorem 11** *For any metric space on at least $(k+1)$ points, no deterministic algorithm for the $k$-server problem with free time can be better than $(k+1)/2$-competitive.*

**Proof:** Recall that $\mathrm{OPT}_i$ is the optimal cost of servicing the requests and ending with the hole at point $i$. Fix $(k+1)$ points in the metric space, and let $d_i$ be the distance from point $i$ to its nearest neighbor. The adversary strategy is simply to request wherever the on-line algorithm's hole is currently located. Thus, if the hole is at point $i$, then the algorithm pays cost at least $d_i$. On the other hand, $\mathrm{OPT}(i)$ increases by at most $2d_i$ because the off-line algorithm could always move its hole from point $i$ to its nearest neighbor and back, at cost $2d_i$. So, if we identify the off-line cost with the *average* of the OPT values, then the ratio of the algorithm's cost to the off-line cost is at least $d_i/(2d_i/(k+1)) = (k+1)/2$. ■

In fact, there exists a (natural) metric space in which one *can* achieve a deterministic ratio of $(k+1)/2$ with free time, in contrast to the lower bound of $k$ [MMS90] in the standard model.

**Theorem 12** *For the metric space of $(k+1)$ equally spaced points on the line, there is a deterministic $k$-server algorithm with free time that achieves competitive ratio $(k+1)/2$.*

**Proof:** Label the points in the space as $0, 1, \ldots, k$ and assume the off-line algorithm begins with its hole at point 0. $\mathrm{OPT}(i)$ is always odd when $i$ is odd and even when $i$ is even. The on-line algorithm places its hole at some point $i$ such that $\mathrm{OPT}(i)$ is a local minimum: i.e., $\mathrm{OPT}(i) < \min(\mathrm{OPT}(i-1), \mathrm{OPT}(i+1))$. When the on-line algorithm's hole is requested, the algorithm pays cost 1 to move it to an adjacent location, and then in its free time it moves to another local minimum. On the other hand, $\mathrm{OPT}(i)$ increases by 2 since it must increase (because it is a local minimum) and cannot change parity. Therefore, the average increase in the OPT values is at least $2/(k+1)$. ■

## 9.4 Achieving Constant Competitive Ratio

We have proven above that there is an $\Omega(\log k)$ lower bound on the competitive ratio of an on-line algorithm with free time for any approximate star space. In this section we show that there exist spaces for which the competitive ratio with free time is constant, and without free time is $\Omega(\log k)$.

**Definition 1** *For function $f()$, the $f$-dumbbell space on $n$ points (assume $n$ is a power of 2) is defined as follows. If $n = 2$ then it consists simply of two at distance 1 apart. Otherwise, it consists of two $f$-dumbbell spaces $M_0, M_1$ each on $n/2$ points, separated from each other by distance $f(n)$. That is, every point in $M_0$ is distance $f(n)$ from every point in $M_1$.*

**Theorem 13** *For $f(n) = 2^n$, the $(k+1)$-point $f$-dumbbell space has randomized competitive ratio at most $c_{k+1} = 2 - 1/(k+1)$ in the free-time model.*

The base case $k = 1$ is clear, so assume inductively that there is a $c_{(k+1)/2}$-competitive algorithm for the $(k+1)/2$-point space. The general algorithm is essentially the same as that in [BKRS92], with appropriate use of free time.

As noted in [BKRS92], we may assume that on any sequence of requests causing the off-line cost of the $(k+1)/2$ point space to increase by $s$, the on-line cost is at most $c_{(k+1)/2} \cdot (s + d_{internal})$, where $d_{internal}$ is the diameter of that space. For $i = 1, 2$, let $\mathrm{OPT}(M_i)$ denote the optimal off-line cost of servicing the requests so far and ending with the hole in space $M_i$. Let us say that initially the hole is in space $M_0$, so initially $\mathrm{OPT}(M_0) = 0$ and $\mathrm{OPT}(M_1) = f(k+1)$.

The algorithm's strategy is simply this. Let $d = f(k+1)$. If the value of $\mathrm{OPT}(M_i)$ increases by $s = d/(k+1)^2$, then move $s/d$ probability mass from $M_i$ to $M_{1-i}$. (If there is already zero probability mass on $M_i$ then do nothing). So, at any point in time, the amount of probability mass on one of the spaces $M_i$ is one of the values $\{0, s/d, 2s/d, \ldots, 1 - s/d, 1\}$. We may do our movement between the two spaces in our free time, so we do not have to pay for it. Our movement schedule ensures that if $\mathrm{OPT}(M_i) = \mathrm{OPT}(M_{1-i}) + d$ then we have zero probability on space $M_i$. Therefore, we may assume that $\mathrm{OPT}(M_i)$ is never constrained by $\mathrm{OPT}(M_{1-i})$.

To calculate our cost on a sequence of requests, we can partition the sequence into intervals in two ways. Partition $P_0$ is a partition of the sequence into intervals that cause $\mathrm{OPT}(M_0)$ to increase by $s$, and $P_1$ is a partition into intervals that cause $\mathrm{OPT}(M_1)$ to increase by $s$. Let us pair up our cost incurred in space $M_0$ inside an interval of $P_0$ in which our probability mass on $M_0$ decreases from $p$ to $p - s/d$ with our cost incurred in space $M_1$ in an interval of $P_1$ in which our probability mass on $M_0$ increases from

$p - s/d$ to $p$. (Our total cost on unpaired intervals is finite.) In such a pair of intervals, the average off-line cost increases by $s$ and our total cost is at most:

$$\begin{aligned} & pc_{(k+1)/2}(s + d_{internal}) + (1 - p + s/d)c_{(k+1)/2}(s + d_{internal}) \\ = \;& c_{(k+1)/2}(1 + s/d)(s + d_{internal}) \\ = \;& s(c_{k+1} - \frac{1}{k+1})(1 + \frac{s}{d} + \frac{d_{internal}}{s} + \frac{d_{internal}}{d}) \\ \leq \;& sc_{k+1} \end{aligned}$$

by our choices of $d_{internal} \ll s \ll d$. ∎

## 9.5 Hints And Free Time

One of the advantages of the free time setting is that it allows a more meaningful incorporation of "hints" about the future into the competitive model. It is well-known that in the standard model an adversary can modify a sequence so that a finite window into the future will not help an algorithm. But in the presence of unlimited free time, a window showing the next request will allow an algorithm to incur no cost. Thus, it is natural to consider hints that have some probability of error.

We now present two models of next-request hints with errors. In the first model, the adversary chooses a fixed request sequence. After presenting an element of the sequence, the adversary gives a "hint" that with probability $p$ is the next element, and with probability $1 - p$ is arbitrary. We refer to this model as the "adversarial request sequence" model.

In the second model the adversary chooses a sequence of hints and presents the appropriate hint at each round. With probability $p$ the actual request is the hint, and with probability $1 - p$ the actual request is arbitrary. We refer to this model as the "adversarial hint sequence" model.

We begin by analyzing the uniform metric space with free time and hints for these models.

**Theorem 14** *In the adversarial request sequence model with hint probability $p$, there is a $(1 + (1 - p)\log k)$-competitive algorithm for the $k$-server problem on the $(k + 1)$-point uniform space. On the other hand, for $p \neq 1$, there remains an $\Omega(\log k)$ lower bound for the adversarial hint sequence model.*

**Proof:** Let the points of the space be labeled $0, \ldots, k$. We consider first the adversarial hint sequence model. The adversary chooses two streams, a hint stream whose requests are all for point 0, and a request stream whose elements are chosen uniformly from $0, \ldots, k$. At each step the adversary passes the algorithm a hint from the hint stream. Then with probability $p$ the true request is the hint, and with probability $1-p$ the true request is the corresponding element from the request stream. The off-line algorithm will always keep its hole away from point 0, and the hints provide no additional information to the algorithm. Thus, the analysis of [FKM+91] (or Theorem 8) shows an $\Omega(\log k)$ lower bound on the performance of the algorithm.

Now we consider the adversarial request sequence model. We modify the marking algorithm of [FKM+91] to take advantage of hints. Recall that at each request to an unmarked vertex, the marking algorithm first marks the vertex and then if no server is present, moves a random unmarked server (chosen uniformly) to the vertex. Once all vertices are marked, the phase ends and all vertices are again unmarked. We modify the algorithm as follows. If the hint is for a marked vertex, we do nothing. Otherwise we move a server to the hint so as to guarantee that the location of the hole is uniformly distributed across the unmarked vertices that are not the hint. (One way to do this is simply to move the server from the previous hint if it so happens that there is no server already at the current hint.) If there is only 1 remaining unmarked vertex, we do not use the hint.

Consider the cost to our algorithm for a single phase. For the $i^{th}$ request of the phase to an unmarked vertex, where $1 \leq i \leq k$, with probability $p$ we pay nothing and with probability $1-p$ we pay $1/(k+1-i)$. Finally, for the $(k+1)^{st}$ request to an unmarked vertex we pay 1. So, the cost to our algorithm is

$$1 + (1-p)\sum_{i=1}^{k} 1/(k+1-i) \approx 1 + (1-p)\log k.$$

So for $p < 1 - 1/\log k$ our cost decreases linearly in the probability that the hints are correct. For hints with a greater probability of correctness, however, the cost remains 1 per phase. ∎

Now consider the case of general $(k+1)$-point metric spaces with hints and free time. The strategy used above can be applied directly to the "Mark" portion of the "Mark and Jump" algorithm discussed in Section 8.3. Thus, hints allow for competitive ratio $O(\log k + (1-p)\log^2 k)$ for general spaces.

**Corollary 15** *There is an algorithm with competitive ratio $O(\log k + (1-p)\log^2 k)$ for general $(k+1)$-point spaces in the adversarial request sequence model with hint probability $p$.*

## 9.6 Bounded Free Time

In this section, we extend our results for unlimited free time to a model in which a limited amount of free time is available after each request. More formally, if an algorithm has free time $F$, the algorithm may perform operations with total cost less than or equal to $F$ between requests, without charge.

We restrict our attention to algorithms that are *lazy* in the sense that they move the hole only when the adversary hits the hole; we do not know whether a loss of generality is involved in this claim (no loss of generality is involved for the traditional model — see [MMS90]). All algorithms given in this paper are lazy, or can be converted to lazy algorithms in a straightforward manner.

Finally, we restrict our attention to metric spaces whose points are nodes of an unweighted graph, and whose distances are given by the shortest path between the points. Let $D$ be the diameter of the space. We show the following theorem:

**Theorem 16**

1. *Any server algorithm with unlimited free time and competitive ratio $r$ can be converted to an algorithm with constant free time and competitive ratio $r \log D$.*
2. *Any server algorithm with unlimited free time can be converted to an algorithm with $\log D$ free time and the same competitive ratio (to within a constant).*

**Proof:**

Before proving the theorem, we require one lemma:

**Lemma 5** *With free time $F$ it is possible to traverse a path of length $L$ with cost bounded by $(4/F)\log(L/F)$.*

**Proof:** We show that after $n$ steps it is possible to be randomly located among the first $nF$ steps of the path. Clearly this is true for $n = 1$. Assume true for some $n$, we will prove it true for $n+1$. If we are located uniformly at random at one of $nF$ locations along the path, break these locations into $n$ batches of $F$ points each. For points in batch $i$, with probability $i/(n+1)$ we jump forward $F$ steps, and with the remaining probability we remain in place.

Assume that $p = 1/nF$ is the probability that we begin at each of the first $nF$ points. The new probability of being at a point in batch 1 is

$$\left(1 - \frac{1}{n+1}\right) p = \frac{n}{n+1} \cdot \frac{1}{nF} = \frac{1}{(n+1)F},$$

which is the correct value. For batch $n+1$, the final probability of being at a particular point is $pn/(n+1) = 1/((n+1)F)$, again the correct value. For batch $i$ where $2 \le i \le n$, the final probability of landing at a point in batch $i$ is given by the probability of starting at that point and remaining there, plus the probability of starting one batch earlier and jumping to the current batch:

$$p\left(1 - \frac{i}{n+1}\right) + p\frac{i-1}{n+1} = p\left(\frac{n+1-i}{n+1} + \frac{i-1}{n+1}\right) = \frac{pn}{n+1} = \frac{1}{(n+1)F}.$$

Thus in $L/F$ steps it is possible to be located at a random location between 1 and $L$. After the first step, the adversary can hit the algorithm with probability no greater than $1/F$. After the second step, the probability is no greater than $1/(2F)$, then $1/(3F)$ and so on. We assume that if the algorithm is hit, it serves the request then moves the server back to its original location, with cost 2. Thus the total cost to the algorithm to this point is bounded by:

$$\begin{aligned}
\text{Total Cost} &\le 2\left(\frac{1}{F} + \frac{1}{2F} + \frac{1}{3F} + \cdots + \frac{1}{(L/F)F}\right) \\
&= \frac{2}{F}\left(\frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{(L/F)}\right) \\
&\le 2\frac{\log(L/F)}{F}.
\end{aligned}$$

The probability can now be coalesced to the last point by following the same procedure in reverse, with additional cost bounded by $(2/F)\log(L/F)$. The total cost is therefore bounded by $(4/F)\log(L/F)$, which completes the lemma. ■

We can now prove the theorem.

**Proof of Theorem:**

For the first part of the theorem, let $F = 1$ in the lemma, giving cost $4\log L$ for a path of length $L$. With free time, the original algorithm must pay at least 1, and our new algorithm will pay no more than $4\log L$. Thus the total cost to our algorithm will be no more than $4\log D$ greater than the cost to the original algorithm, which gives the first claim of the theorem.

For the second claim, we set $F = \log D$. In order for our algorithm to traverse a path of length $D$ it must pay at most $(4/\log D)(\log(D/\log D)) \le 4$. This completes the theorem. ■

## 9.7 Free Time For Command Processing

In this section we incorporate free time into a simple model of command processing. Typically competitive analysis is applied to sequences that are computer generated at a small timescale; for instance, deciding what to cache [MMS90, FKM+91], moving disk heads to serve requests for disk blocks [CKPV90], or deciding whether a process should spin or block [KMMO94]. Here, we consider the problem of interacting with a user. In our model, a user presents a request which the algorithm must service, waits for some amount of time, then presents the next request. The amount of free time between requests is limited but unknown to the algorithm. Unlike in the previous sections, we compare the algorithm to an off-line algorithm who knows what the command will be *and* has access to the free time; the off-line optimal strategy is therefore to use all the free time processing the next command.

We focus on a simple situation. The user may choose from two possible commands, $C_A$ and $C_B$. Processing command $C_A$ takes time $A$, and processing command $C_B$ takes time $B$. We assume that $A > B$; as a concrete example, $C_A$ might be the command "latex file.tex" and $C_B$ might be "ls." We will first present the optimal strategy when the amount of free time is known to the algorithm, and then extend to the case in which the amount of free time is not known.

Assume there are $t$ units of free time, where $t < B$. The algorithm must choose to spend $a$ units of time processing $C_A$ and $b$ units of time working on $C_B$ such that $a+b = t$. If $C_A$ is chosen the algorithm's cost is $A - a$ and the off-line cost is $A - t$. Likewise, if $C_B$ is chosen, the algorithm's cost is $B - b$ and the off-line cost is $B - t$. The competitive ratio is therefore:

$$\text{C.R.} = \max\left\{\frac{A-a}{A-t}, \frac{B-b}{B-t}\right\}.$$

Setting these factors equal, we see that the appropriate split of time between $a$ and $b$ is given by

$$\frac{a}{b} = \frac{B-t}{A-t}.$$

This gives the optimal strategy for the algorithm if the free time is known in advance. One way to view this is that if $A \gg B$ then one should put most of one's effort into preprocessing $B$ since that is where one can make the most impact.

We assume next that the algorithm does not know how much free time there will be. As each additional unit of time becomes available, the algorithm must decide how to split the single unit between $C_A$ and $C_B$. During the $i^{th}$ unit of free time let $a_i$ be the amount of time spent working on $C_A$ and $b_i$ be the amount of time spend working on $C_B$, $a_i + b_i = 1$. From the result above, it is clear than as $t \to B$, the algorithm should eventually spend all its free time working on $C_B$ as the adversary will be able to service

$C_B$ with no cost. But earlier, the algorithm will be obligated to spend some time working on $C_A$ in order to maintain the optimal competitive ratio. This motivates the following question: for how long can the algorithm perform optimally?

Assuming that the algorithm has performed optimally for $t$ units of free time, it must be the case that

$$\frac{\sum_{i=1}^{t} a_i}{\sum_{i=1}^{t} b_i} = \frac{B-t}{A-t}.$$

If $t \to B$ the algorithm must spend all of the next unit of time working on $C_B$, which will result in a new ratio of

$$\frac{\sum_{i=1}^{t} a_i}{1+\sum_{i=1}^{t} b_i} = \frac{t(B-t)}{-t^2+t(A-2)+A+B}. \tag{9.1}$$

This ratio should be $(B-(t+1))/(A-(t+1))$, so in order to determine when the algorithm is no longer capable of "keeping up" with the changing ratio $a/b$, we find $t$ such that equation 9.1 is equal to this new ratio.

Solving, and using the simplification $A+B+1 \approx A+B$, we see that

$$t = \frac{A+B-\sqrt{A(A+B)}}{2}.$$

For $B \ll A$, we can make the approximation $\sqrt{A(A+B)} \approx A$, which gives $t = B/2$. Thus, for large $A$, we can remain optimal until our free time reaches half the time to compute the smallest command. For larger $B$, however, we can do substantially better. Assume that for some constant $\epsilon$, we have $B = A\epsilon$. Then $\sqrt{A(A-B)} = \sqrt{A^2(1-\epsilon)} = A\sqrt{1-\epsilon} \approx A(1-\epsilon/2-\epsilon^2/8)$, which results in $t = (A+B-A(1-\epsilon/2-\epsilon^2/8))/2 = (B+A\epsilon/2+A\epsilon^2/8)/2 = (B+B/2+B\epsilon/8)/2 \approx 3B/4$. Thus, when $B$ grows to be a constant fraction of $A$, we can remain optimal until our free time approaches three quarters of the total time to compute $C_B$.

# Chapter 10

# Conclusion

*And so ends my catechism.*

— *William Shakespeare, "Henry IV, Part I"*

This thesis has two technical parts. Part I considers systems for disk prefetching and disk cache management in the presence of information about upcoming application I/O. Part II considers online algorithms, which must respond to each request in a sequence without knowing what the next request will be. Specifically, I consider an extension to the traditional cache management problem called weighted caching, in which certain pieces of cached data are more expensive to re-load than others; I also study an extension to the traditional online model in which the algorithm responding to requests has free time between finishing one request and receiving the next, during which it can prepare without charge.

## 10.1 Systems Conclusions

This section contains a summary of the results of the systems part of the thesis. The technical data is presented in two chapters representing standalone processes (chapter 5), and multiple processes running simultaneously (chapter 6). In each case I presented trace-driven simulations from actual and synthetic data, and then distilled these experiments into a small number of "lessons learned" in the course of doing the work. To conclude, I gather these lessons together and present them here with some high-level discussion.

I begin with four lessons that are particularly evident in the single-process case. The first two summarize our conclusion that FORESTALL's dynamic estimates of disk load perform well under both heavy and light load. Static approaches, such as TIP2's conservative prefetching or AGGRESSIVE's aggressive prefetching, tend to perform well in some situations, but poorly in others. The final two lessons summarize locality issues resulting from queueing and eviction respectively. These four lessons are described in detail in Section 5.4.

**Lesson 1:** *Leaving a constrained disk idle leads to additional stall.*

**Lesson 2:** *Submitting an I/O requires* $T_{\text{driver}}$ *computational overhead.*

**Lesson 3:** *Deeper disk queues yield lower average disk service times.*

**Lesson 4:** *Eviction decisions impact locality of "re-fetched" data.*

The next six lessons occur primarily in multi-process situations. The first is the single most important conclusion of Chapter 6: LRU-SP's rate-based allocation scheme does not necessarily put buffers where they are most useful. The next two lessons show that the appropriate division of resources between hinted and unhinted streams is different for constrained versus unconstrained disks. And the final three lessons concern short hint queues, thrashing, and prioritization schemes respectively. All these lessons are described in detail in Section 6.5

**Lesson 5:** *Access rate is not a good predictor of caching value.*

**Lesson 6:** *On unconstrained disks, hinted blocks can always be prefetched in time so caching them is not as important as caching for unhinted accesses.*

**Lesson 7:** *On constrained disks, hinted blocks that are ejected cannot be re-fetched without stall so caching them is as important as caching for unhinted accesses.*

**Lesson 8:** *Constraint-aware prefetching only reasons about* known *constraints.*

**Lesson 9:** *Over-aggressive prefetching may result in eviction of prefetched but unread data.*

**Lesson 10:** *Batching for efficient I/O may be defeated by prioritization schemes.*

### 10.1.1 Discussion

At the highest level, there is a single principle that implies many of the lessons above: performance always improved when we incorporated application knowledge more deeply into the prefetching and allocation decisions. For example:

- Chapter 5 shows that prefetching policies that statically fetch blocks either conservatively or aggressively without regard to the particular application and environment do not perform as well as dynamic policies that adapt to the situation. Lesson 1 shows that being conservative all the time may result in additional stall, while Lesson 2 shows that being too aggressive may result in additional overhead from unnecessary I/O's.

- As Lesson 5 shows, allocation policies that dedicate a fraction of the cache to a particular purpose independent of the degree of re-use do not perform as well as policies that allocate the cache dynamically based on application re-use patterns. For instance, unless a process is moving extremely slowly, LRU-SP will give a significant fraction of the cache to that process whether or not it can use the space. COST-BENEFIT, on the other hand, will give the process a much larger share of the cache if it demonstrates that more buffers will improve latency, and a much smaller fraction if the process shows no re-use.

- Lessons 6 and 7 show that prefetching and cache management policies must be aware of the I/O load presented by the application — as the load grows, the policy must be able to switch modes from aggressive caching to aggressive prefetching.

The remaining lessons can all be summarized by a similar high-level principle: performance improvements result from experimentally identifying and addressing inaccuracies in the system model. In some cases, the correct approach may be to improve the system model (for instance, extending AGGRESSIVE to FORESTALL requires incorporating a notion of $T_{\mathrm{driver}}$). In other cases, however, such an extension may not be feasible and it may be necessary to address the problem elsewhere. For instance, disk queueing algorithms such as CSCAN do not incorporate an accurate model of disk geometry, but nonetheless address the problem of non-constant disk access times effectively.

- Lessons 3, 4, and 10 all follow from the observation that disk access time is not constant. Most immediately, Lesson 3 shows that deeper disk queues tend to improve disk access times. Lesson 4, regarding re-fetch locality, shows that even eviction decisions must take disk access times into consideration. And Lesson 10 shows that queueing benefits may be impacted by I/O priorities.

  The current approaches (queueing, batching, and not promoting prefetches that become demand reads to a higher priority level) are effective for all the applications in the suite. It is possible, however, that future work will uncover the need for a more complex model of disk geometry in the system model.

- Lesson 8 concerns hints that trickle in over time. As the hint queue drains, algorithms may under-estimate the I/O load on a disk because upcoming constraints have not yet been made visible.

- Lesson 9 concerns buffers that are evicted before being read. However, in some situations the behavior is appropriate — a set of blocks might have been prefetched correctly, but due to changing system conditions, those blocks should now be evicted and re-fetched before they are read. As described in Section 4.8.12, there are a number of safeguards in the existing system to protect against unnecessary early eviction.

### 10.1.2 Future Systems Work

Over a period of about a year, the Parallel Data Lab at CMU, and groups elsewhere (especially at the University of Washington in Seattle), have been evaluating algorithms for the informed prefetching and caching problem. Part I contains two kinds of results. First, I compare various algorithms in single-process and multi-process environments, and draw conclusions about their relative performance. Second, I study the particular implementation details of the stronger algorithms and suggest which elements are likely to be important and which do not contribute significantly, which particular forms of the algorithms are most effective, and so on. This work was designed to explore and prune the space of possible implementations, and the next step will be to generate and test a kernel-level version of the best algorithm. Thus, the largest piece of future work is part of a long-range plan into which this work fits.

Next, the simulator is general and extensible, and would be an appropriate tool for other explorations in this area. For instance, as discussed in Section 2.3, there are several approaches to generating hints automatically, ranging from compiler-based approaches to speculative execution. One of the issues in automatic hint generation is dealing with incorrect hints. There would be significant benefits to evaluating tracking algorithms for this problem via simulation rather than implementation, especially since the search space for possible algorithms is so wide. Implementing such algorithms on top of the existing code would be straightforward. Similarly, the simulator represents a general-purpose tool for studying different aspects of cache management, and could easily be extended to examine weighted variants of the problem, networked data sources, algorithms that are aware of disk non-linearities, and so on.

## 10.2 Theory Conclusions

The theory part of the thesis makes two main contributions. The first, presented in Chapter 8, is an analysis of the weighted caching problem for metrical task systems, or for $k$-server problems on $(k+1)$-point spaces. We give randomized algorithms and lower bounds that are nearly tight, establishing that the competitive ratio for the problem is between $\Omega(\log k)$ and $O(\log^2 k)$. The second contribution, presented in Chapter 9, is an extension to the traditional online model to allow free time between servicing a request and receiving the next request, in which work may be performed without cost. We give a number of results in this model. Most importantly, we show that any algorithm with free time may be converted to a weighted caching problem without free time, and that therefore the results of Chapter 8 give a randomized $O(\log^2 k)$-competitive algorithm for any metric space with free time. A complete summary of the results of this chapter appears in Section 9.1.

### 10.2.1 Future Theory Work

Results of Bartal, Blum, Burch and myself [BBBT96] give a polylog-competitive algorithm for metrical task systems on any metric space, extending our results for weighted-cache spaces with a slight increase in the competitive factor (from $O(\log^2 k)$ to $O(\log^6 k)$). This essentially closes the door on extensions to other metrical task system spaces. However, there is an interesting orthogonal direction for extensions: weighted caching with multiple holes. Our results apply to metrical task systems on weighted-cache spaces, or, according to Theorem 4, the $k$-server problem for $(k+1)$-point weighted-cache spaces. The results of [BBBT96] can be applied to generate an algorithm with polylog competitive factor for spaces with polylog($n$) holes. Beyond this, (even, say, for a space of $k + \sqrt{k} \ll 2k$ points) the question is open. Discussions with Yair Bartal, Amos Fiat, Avrim Blum, Adi Rosen and Neal Young suggest that the problem of weighed caching with only two weights can be solved in general (*i.e.*, there is a algorithm competitive in the number of servers $k$, independent of the size of the space $n$), though as yet I do not believe anybody has written up the result. Unfortunately, as new weights are added the competitive ratio increases geometrically so this does not give a good bound for more general weighted caching spaces. The Marking Algorithm of [FKM+91] is shown to be competitive for unweighted caching with an arbitrary number of holes, but the techniques of that paper do not appear to generalize in a straightforward manner. However, in the same way that techniques from weighted caching extended in the metrical task system domain to techniques for arbitrary spaces, it seems reasonable to believe that weighted caching in the $k$-server domain is a logical place to focus efforts.

## 10.3 Theory and Practice

I chose to present the systems and theory parts of the thesis as two distinct pieces to avoid the temptation to create connections where they don't really exist. The two pieces of work were performed as standalone research. Nonetheless, there are similarities in the two parts, along with some subtle differences. This section discusses how the two parts compare, and then offers some thoughts on bridging the gap in the future.

### 10.3.1 Online Versus Offline Problems

Both parts of the thesis are concerned with cache management. The high-level difference is that the systems part deals with what the theory part would refer to as the *offline problem*, in which the sequence of future accesses is known to the algorithm (this sequence is simply the disclosures generated by the application). However, although the systems problem is offline by nature, there are online issues that arise. First, TIPTOE in the presence of multiple processes may be given complete and correct disclosures for each, but it will not know the interleaving of the requests. In fact, this exact model has been studied under competitive analysis by Barve, Grove and Vitter *et al.* [BGV95]. Furthermore, even for a single process, there are again online elements of the problem. For instance, if the hints trickle in over time, as in the SPHINX trace, the system might only see a finite window into the future — it is a folklore online algorithms result that no finite window will improve the competitive ratio (the result relies on the worse-case nature of the competitive model). If, on the other hand, we are given all the hints but they are incorrect with a certain probability, we can imagine a model such as the hinting model of Section 9.5, in which an algorithm must decide how much to rely on hints that might be erroneous.

Finally, there are also useful results that can be proven for offline problems in a fixed analytical model. While the general offline $k$-server problem is not known to have a poly-time solution, the weighted-caching problem can be solved using a network-flow algorithm. However, once we begin to seek an algorithm with provable properties to solve the problem TIPTOE is solving, we encounter the second major difference between the systems and theory parts of the thesis.

### 10.3.2 Practical Issues and System Models

To recap, the first major difference is that TIPTOE has knowledge about future accesses, while the Mark-And-Jump algorithm does not. The second major difference is the traditional difference between practical systems problems and their analytical counterparts: there are many critical details in the systems problem that are not reflected in the analytical model.

Building an offline algorithm based on the $k$-server model has a flaw: servers are charged for motion, so essentially the metric is amount of time spent doing I/O. This metric may be appropriate for some problems, but is not appropriate for TIPTOE's domain: we wish to uncover opportunities for latency hiding via prefetching, and a total

I/O metric will not allow us to do so. As described in Section 4.1.1, a more appropriate metric is the increase in I/O service time (*i.e.*, the increase in stall, plus the increase in other overhead incurred by the I/O subsystem). A model to address this difficulty using the competitive framework is given by Cao, Karlin *et al.* [CFKL95b], and by Kimbrel and Karlin in the multi-disk case [KK96b]. Their model counts I/O stall time rather than simply I/O time, but still does not take into account other components of I/O service time (*e.g.*, $T_{\text{driver}}$). It is not known whether their results can be extended to a model with positive $T_{\text{driver}}$. The discussion of TIPTOE does not include a theoretical analysis because my personal experience suggested that an accurate analytical model would require a great deal of effort to create.

In fact, much of the work of developing TIPTOE focused on understanding the impact of disk non-linearities, queueing, re-fetch locality, and so on; the standard theoretical models all assume constant disk access times. Similarly, it took significant effort to generate an effective set of estimators of the consumption rate of a process, the average I/O rate of a process, and so on; these are usually assumed to be parameters in the theoretical model. So to summarize, a simple analytical model may give useful insights into algorithms for the problem, but it is dangerous to assume that performance proofs within the model will also hold in the real world.

### 10.3.3 Weighted Caching and Different Disk Loads

The third major difference between the systems and theory parts of the thesis is the precise problem being studied. TIPTOE does not address weighted caching precisely, although there are connections. When we began the systems work, we believed that two important extensions to TIP2 would be required, and we expected that the approaches taken in each case would be similar. The first problem was to incorporate disk layout information, and to favor an overloaded disk over a lightly-loaded one. TIPTOE addresses this problem by maintaining per-disk estimates of overload. The second problem was to model disks with different characteristics, either slow disks and fast disks, or local disks versus servers, or servers with different levels of background load. This problem is more similar to weighted caching, and it is not straightforward to extend TIPTOE to address it. David Rochberg is currently working on *Remote TIP*, a project evaluating architectures for using disclosures to improve I/O performance from networked file servers.

### 10.3.4 Future Connections between Theory and Practice

When I began this thesis, I hoped that it would be possible to incorporate algorithms for weighted caching into the system model described in Chapter 3. I hoped there would be crosstalk between the theoretical results and the practical algorithms. It still seems that there is substantial promise for this approach, whether the algorithms are the types of randomized algorithms described here or their simpler deterministic counterparts such as the BALANCE algorithm described by Chrobak *et al.* [CKPV90]. However, within the scope of this thesis, it has not been possible to perform such an experiment.

# Appendix A

# Proof of Folklore Theorems

**Theorem 3**

*For any metric space and any fixed $\epsilon > 0$, given an $r$-competitive algorithm for the metrical task system problem with $\epsilon$-bounded, elementary task vectors, it is possible to construct an algorithm for the general metrical task system problem with competitive ratio $(1+\epsilon)r$.*

**Proof:**

Let $\sigma$ be a sequence of arbitrary (not necessarily $\epsilon$-bounded elementary) task vectors. First, we will show how to construct a subsequence of elementary task vectors for each single task vector of $\sigma$, and will concatenate the resulting subsequences into a new sequence of elementary task vectors $\tau$. Next, we will show how the behavior of an $r$-competitive algorithm $A$ for sequences of elementary task vectors, operating on sequence $\tau$, can be used to induce a new algorithm $B$ for the original sequence $\sigma$. Finally, we will show that $B(\sigma) \leq (1+\epsilon)A(\tau)$ and $\mathrm{OPT}(\tau) \leq \mathrm{OPT}(\sigma)$.

An individual task vector $v$ of $\sigma$ can be converted into a subsequence of elementary task vectors of $\tau$ as follows. Let $v = (\delta_1, \delta_2, \ldots, \delta_n)$ be an arbitrary task vector, and let $\delta$ be some small value to be determined later.

while some $\delta_i > 0$ do {
    /* begin next stripe */
    for $j \leftarrow 1$ to $n$
        if $(\delta_j > 0)$ {
            output task $(\overbrace{0, \ldots, 0}^{j-1}, min(\delta_j, \delta), 0 \ldots 0)$
            $\delta_j \leftarrow max(0, \delta_j - \delta)$

```
        }
}
```

This construction shows how to create $\tau$ from $\sigma$. Algorithm $B$ on $\sigma$ works as follows. It begins by initializing a copy of algorithm $A$, and maintains the invariant that the state of $B$ after processing some vector $v$ is the state of $A$ after processing the subsequence of elementary task vectors corresponding to $v$. When $B$ is presented with $v$, it creates a subsequence of elementary task vectors according to the construction above, and then passes the resulting vectors to algorithm $A$ one at a time. $A$ begins in some state $s_0$, and then passes through some set of states $S$ in the course of processing the elementary task vectors, and ends in some final state $s_2$. Each of these states will have some cost in the original vector $v = (\delta_1, \delta_2, \ldots, \delta_n)$; let state $s_1$ be the state of $S$ with lowest cost: $s_1 = argmin_{s \in S}\{\delta_j\}$. Algorithm $B$ begins in state $s_0$, immediately jumps to state $s_1$ to process $v$, and finally jumps to state $s_2$ to remain in correspondence with $A$.

Having defined $\tau$ and algorithm $B$, we must show that $B(\sigma) \leq (1+\epsilon)A(\tau)$ and $\mathrm{OPT}(\tau) \leq \mathrm{OPT}(\sigma)$. The first requires a proof, the other is trivial.

The first inequality states $B(\sigma) \leq (1+\epsilon)A(\tau)$. Consider some $v = (\delta_1, \delta_2, \ldots, \delta_n)$, a task vector of $\sigma$. According to the construction above, $\tau$ will contain a subsequence of elementary task vectors corresponding to $v$; we call this subsequence $V$. The construction breaks the subsequence into a number of logical units called "stripes," each stripe consisting of up to $n$ elementary task vectors, and no two vectors in a stripe having non-zero values for the same state. We denote the stripes STRIPE1, STRIPE2,....

In the course of processing $V$, say that algorithm $A$ changes state $k$ times, paying distance $d_i$ on the $i^{th}$ state change. Define $n_1 = \lfloor \delta_{s_1}/\delta \rfloor$. During processing of STRIPE$j$ for $j \leq n_1$, $A$ must either move, paying some distance, or must pay cost $\delta$ by the definition of $s_1$. The total cost to algorithm $A$ over $V$ may therefore be lower bounded by $\mathrm{Cost}(A, V) \geq \sum_{j=1}^{k} d_j + (n_1 - k)\delta$. Let $d_{\min}$ be the smallest distance in the space, and choose $\delta < \epsilon d_{\min}/2$. Additionally, choose $\delta < \epsilon\delta_j/2$ for all $j$, so $n_1\delta > \delta_{s_1}(1-\epsilon/2)$. Finally, let $\delta \leq \epsilon$ to guarantee that the resulting task vectors are $\epsilon$-bounded. Then

$$
\begin{aligned}
\mathrm{Cost}(A, V) &\geq (1-\epsilon/2)\sum_{j=1}^{k} d_j + \epsilon k d_{\min}/2 + (n_1 - k)\delta \\
&\geq (1-\epsilon/2)\sum_{j=1}^{k} d_j + n_1\delta \\
&\geq (1-\epsilon/2)\sum_{j=1}^{k} d_j + (1-\epsilon/2)\delta_{s_1}
\end{aligned}
$$

Finally, note that $A$ must travel from $s_0$ via $s_1$ to $s_2$, so by the triangle inequality the total distance must be at least $d_{s_0,s_1} + d_{s_1,s_2}$. So we can write the final lower bound on the cost of algorithm $A$ as follows:

$$\mathrm{Cost}(A, V) \geq (1 - \epsilon/2)(d_{s_0,s_1} + d_{s_1,s_2} + \delta_{s_1}).$$

Recall that $B$ begins servicing $v$ in $s_0$, jumps to $s_1$ to service the vector, and finally jumps to $s_2$. So the cost to $B$ for servicing $v$ is given by $\mathrm{Cost}(B, v) = d_{s_0,s_1} + \delta_{s_1} + d_{s_1,s_2}$. Thus,

$$(1 + \epsilon)\mathrm{Cost}(A, V) \geq \mathrm{Cost}(B, v),$$

$$B(\sigma) \leq (1 + \epsilon)A(\tau).$$

The second inequality we must show states: $\mathrm{OPT}(\tau) \leq \mathrm{OPT}(\sigma)$. We will fix an optimal solution for $\sigma$ and use it to construct a solution for $\tau$ with the same cost. If the optimal solution for $\sigma$ jumps to state $j$ to serve $\sigma_i$, the solution for $\tau$ will jump to state $j$ to serve the subsequence of elementary vectors in $\tau$ corresponding to $\sigma_i$. This solution for $\tau$ will incur the same movement costs and task vector costs as the optimal algorithm for $\sigma$, by the construction of $\tau$.

Finally, by the assumption of competitiveness, we know that $A(\tau) \leq r\mathrm{OPT}(\tau) + c$, since $\tau$ consists of $\epsilon$-bounded, elementary task vectors.

Thus, we conclude:

$$\begin{aligned} B(\sigma) &\leq (1 + \epsilon)A(\tau) \\ &\leq (1 + \epsilon)(r\mathrm{OPT}(\tau) + c) \\ &\leq (1 + \epsilon)r\mathrm{OPT}(\sigma) + (1 + \epsilon)c \end{aligned}$$

■

**Theorem 4**

*Consider a metrical task system on a metric space $M$ of $k + 1$ points, and the corresponding k-server problem on a $(k+1)$-point space. Given an algorithm $A$ that solves the MTS with competitive ratio $r$, it is possible to construct an algorithm $B$ for the k-server problem with competitive ratio $r$. Likewise, given an algorithm $B$ with competitive ratio $r$ for the k-server problem, it is possible to construct an algorithm $A$ with competitive ratio $7r$ for the MTS.*

**Proof:**

We now use Theorem 3 to complete the proof of Theorem 4. Given an algorithm $A$ for the MTS, it is trivial to create an algorithm $B$ for the $k$-server problem: whenever a request for point $i$ arrives at the $k$-server algorithm, it simply creates a task vector with value 0 for all states $j \neq i$ and value $\infty$ for state $i$. Algorithm $B$ then moves its hole to the new state of algorithm $A$. Let $\sigma$ be the induced sequence of task vectors presented to algorithm $A$ and $\tau$ be the original sequence of points sent to algorithm $B$.

Consider some optimal offline solution for $\tau$. It is possible to construct a solution to $\sigma$ in the MTS domain by always moving to the state corresponding to the hole of $\mathrm{OPT}(\tau)$. This solution to $\sigma$ will never pay local costs, and will pay movement costs equal to the movement costs of $\mathrm{OPT}(\tau)$, so we have $\mathrm{OPT}(\sigma) \leq \mathrm{OPT}(\tau)$.

Algorithm $B$ will incur exactly the same cost as algorithm $A$: $B(\tau) = A(\sigma)$. By the competitiveness of $A$, we have $A(\sigma) \leq \mathrm{OPT}(\sigma) + c_1$. And from the previous paragraph we have $\mathrm{OPT}(\sigma) \leq \mathrm{OPT}(\tau)$. Therefore we conclude

$$B(\tau) = A(\sigma) \leq r\mathrm{OPT}(\sigma) + c_1 \leq r\mathrm{OPT}(\tau) + c_1.$$

It remains to show the other direction. We assume we have an algorithm $B$ for the $k$-server problem, and use it to create an algorithm $A$ for the MTS. For each point $j$ let $w_j$ be the distance to $j$'s nearest neighbor: $w_j \stackrel{def}{=} \min_{i \neq j}\{d_{ij}\}$, and let $w_{\max} = \max_j\{w_j\}$ and $w_{\min} = \min_j\{w_j\}$. Let $\sigma$ be an $\epsilon w_{\min}$-bounded sequence of task vectors, where the value of $\epsilon$ is to be determined later. Let local$(j)$ be a per-point real variable initialized to 0. From $\sigma$ we construct $\tau$ according to the following pseudo-code:

```
foreach σ_i = v = (v_1, ..., v_{k+1})  {
    for j ← 1 to (k + 1)  {
        increment local(j) by v_j
        if (local(j) > w_j)  {
            issue a request for point j
            set local(j) to 0
        }
    }
}
```

We break $B$'s total cost on $\sigma$ into two pieces, the *local* and *movement* costs. Any costs that result from movement between states are considered to be part of *movement_cost*$(\sigma)$, and all remaining costs (those associated with servicing a task in a particular state) are part of *local_cost*$(\sigma)$. Finally for convenience, define $c_1 = (1+\epsilon)w_{\max}$.

**Lemma 6** *local_cost*$(\sigma) \leq (1+\epsilon)$*movement_cost*$(\sigma) + c_1$

**Proof:**

At any point algorithm $B$ moves to some new state $j$, algorithm $A$ does likewise. Algorithm $A$ may incur local cost until local$(j) \geq w_j$, at which point a request will be generated for point $j$ to algorithm $B$, and algorithm $A$ will move to another state. Since task vectors are $\epsilon w_{\min}$-bounded, the local cost is no more than $w_j + \epsilon w_{\min} \leq w_j(1+\epsilon)$, and the cost of moving from state $j$ is at least $w_j$. Over the entire sequence $\sigma$, this will be true for all states algorithm $A$ enters except the last one, which may incur an additional $(1+\epsilon)w_j$ local cost with no offsetting movement. The total local cost is therefore no more than $(1+\epsilon)\mathit{movement\_cost}(\sigma) + c_1$, which completes the lemma.

■

**Lemma 7** $A(\sigma) \leq (2+\epsilon)B(\tau) + c_1$

**Proof:**

By Lemma 6, we have

$$A(\sigma) = \mathit{local\_cost}(\sigma) + \mathit{movement\_cost}(\sigma) \leq (2+\epsilon)\mathit{movement\_cost}(\sigma) + c_1.$$

Noting that the movement costs of $A$ and $B$ are equivalent, and that this value is exactly $B(\tau)$, we have

$$A(\sigma) \leq (2+\epsilon)\mathit{movement\_cost}(\sigma) + c_1 = (2+\epsilon)B(\tau) + c_1.$$

■

This completes the relation between online cost in the MTS and $k$-server domains. We now consider the offline cost.

**Lemma 8** $\mathit{OPT}(\tau) \leq 3\mathit{OPT}(\sigma)$

**Proof:**

We construct a solution for $\tau$, which we will call OPT$'(\tau)$ or simply OPT$'$, from some optimal solution to $\sigma$, OPT$(\sigma)$. OPT$'$ behaves as follows. Its hole tracks the state of OPT$(\sigma)$ with the sole exception that, when a request arrives in $\tau$ for the hole of OPT$'$, OPT$(\sigma)$ may not move, but OPT$'$ moves the hole to its nearest neighbor, serves the request, and moves the hole back before the next request arrives. We must show that the cost incurred by OPT$'$ is not much more than the cost incurred by OPT$(\sigma)$.

Each unit of cost incurred by OPT$(\sigma)$ will be tripled and deposited at some point of the space so as to completely cover the costs of OPT$'$ (more specifically, movement costs will be tripled, but local costs will be doubled). This will complete the lemma. The rules for depositing cost are the following. Whenever OPT($\sigma$) incurs local cost, twice that cost is deposited on the point of the metric space in which it was incurred. Whenever OPT($\sigma$) moves, one unit of the movement cost is used to pay for the corresponding move in OPT$'$, and the other two units are deposited at the destination.

Whenever a request $\tau_i \in \tau$ arrives for point $j$, the current hole of OPT$'$, the hole is moved to the nearest neighbor and back. There are two cases:

1. If OPT($\sigma$) entered state $j$ most recently *after* the prior request to point $j$ in $\tau$ then the movement will have deposited twice the cost to move to state $j$ from some other state — this will be enough to cover the cost in OPT$'$ of moving to the nearest neighbor and back.

2. If OPT($\sigma$) entered state $j$ most recently *on or before* the prior request to point $j$ in $\tau$ then by definition the local cost to OPT($\sigma$) will be at least $w_j$. Therefore twice that amount will have been deposited at point $j$, which is enough to move to the nearest neighbor and back.

This shows that, if the costs incurred by OPT($\sigma$) are tripled, they outweigh the costs incurred by OPT$'$.

■

We have shown the following sequence of inequalities:

$$\begin{aligned} A(\sigma) &\leq (2+\epsilon)B(\tau)+c_1 \\ B(\tau) &\leq r\mathrm{OPT}(\tau)+c_2 \\ \mathrm{OPT}(\tau) &\leq 3\mathrm{OPT}(\sigma) \end{aligned}$$

We can chain these inequalities together as follows:

$$\begin{aligned} A(\sigma) &\leq (2+\epsilon)B(\tau)+c_1 \\ &\leq (2+\epsilon)(r\mathrm{OPT}(\tau)+c_2)+c_1 \\ &\leq (2+\epsilon)(3r\mathrm{OPT}(\sigma)+c_2)+(1+\epsilon)c_1 \\ &\leq 3r(2+\epsilon)\mathrm{OPT}(\sigma)+c_3 \end{aligned}$$

Finally, by Theorem 3, we have an algorithm for the original non-$\epsilon$-bounded task sequence with competitive ratio no greater than $3r(2+\epsilon)(1+\epsilon) < 7r$ for appropriate choice of $\epsilon$.

■

# Bibliography


[ADU71] Alfred V. Aho, Peter J. Denning, and Jeffrey D. Ullman. Principles of optimal page replacement. *Journal of the ACM*, 18(1):80–93, January 1971. (p 39)

[Bar96] Yair Bartal. Probabalistic approximation of metric spaces and its algorithmic applications. In *Proceedings of the 37th Annual IEEE Symposium on Foundations of Computer Science*, pages 183–193, October, 1996. (p 149)

[BBBT96] Yair Bartal, Avrim Blum, Carl Burch, and Andrew Tomkins. A polylog(n)-competitive algorithm for metrical task systems. In *submitted*, 1996. (pp 23, 137, 147, 149, 179)

[BBK+90] S. Ben-David, A. Borodin, R. Karp, G. Tardos, and A. Wigderson. On the power of randomization in online algorithms. In *ACM Symposium on Theory of Computing*, pages 379–386, 1990. (p 139)

[BC91] J. L. Baer and T. F. Chen. An effective on-chip preloading scheme to reduce data access penalty. *Supercomputing '91*, 1991. (p 38)

[Bel66] L.A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5:78–101, 1966. (p 39)

[BGV95] Rakesh Barve, Edward F. Grove, and Jeffrey Scott Vitter. Application-controlled paging for a shared cache. In *36th Annual Symposium on Foundations of Computer Science*, pages 204–213, Milwaukee, Wisconsin, October 23–25 1995. IEEE. (pp 39, 180)

[BKRS92] A. Blum, H.J. Karloff, Y. Rabani, and M. Saks. A decomposition theorem and bounds for randomized server problems. In *Proceedings of the 33rd Annual IEEE Symposium on Foundations of Computer Science*, pages 197–207, 1992. (pp 23, 143, 145, 147, 149, 151, 164, 167)

[BLS87] A. Borodin, N. Linial, and M. Saks. An optimal online algorithm for metrical task systems. In *19th ACM Symposium on Theory of Computing*, pages 373–382, 1987. (p 141)

[BLS92] A. Borodin, N. Linial, and M. Saks. An optimal online algorithm for metrical task systems. *JACM*, 39(4):745–763, 1992. (pp 146, 147)

[Cao96] Pei Cao. *Application-Controlled File Caching and Prefetching.* PhD thesis, Princeton University, 1996. (pp 28, 29, 32, 33, 59, 65)

[CB92] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, October 1992. (p 38)

[CD85] H.T. Chou and D.J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the 11th International Conference on Very Large Data Bases*, pages 127–141, 1985. (p 36)

[CDKK85] H.-T. Chou, David J. Dewitt, Randy H. Katz, and Anthony C. Klug. Design and implementation of the wisconsin storage system, January 1985. (p 36)

[CFKL95a] P. Cao, E.W. Felten, A. Karlin, and K. Li. Implementation and performance of integrated application-controlled caching, prefetching and disk scheduling. Technical Report TR-CS95-493, Princeton University, 1995. (p 33)

[CFKL95b] P. Cao, E.W. Felten, A. Karlin, and K. Li. A study of integrated prefetching and caching strategies. In *Proceedings of the ACM SIGMETRICS*, May, 1995. (pp 18, 20, 30, 33, 59, 63, 82, 99, 131, 181)

[CFL94a] P. Cao, E.W. Felten, and K. Li. Application-controlled file caching policies. In *1994 Usenix Summer Technical Conference*, pages 171–182, June, 1994. (pp 20, 32, 33, 65)

[CFL94b] P. Cao, E.W. Felten, and K. Li. Implementation and performance of application-controlled file caching. In *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation, Monterey, CA*, pages 165–178, November, 1994. (pp 18, 28, 33, 65)

[CKPV90] M. Chrobak, H. Karloff, T. Payne, and S. Vishwanathan. New results on server problems. In *First Annual ACM-SIAM Symposium On Discrete Algorithms*, pages 290–300, 1990. (pp 146, 172, 182)

[CKV93] K. Curewitz, P. Krishnan, and J.S. Vitter. Practical prefetching via data compression. In *Proceedings of the 1993 ACM Conference on Management of Data (SIGMOD)*, pages 257–266, May, 1993. (p 35)

[CL91] Marek Chrobak and Lawrence Larmore. An optimal on-line algorithm for $k$ servers on trees. *SIAM J. Computing*, 20(1):144–148, 1991. (p 146)

[CP90] Peter M. Chen and David A. Patterson. Maximizing performance in a striped disk array. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 322–331. IEEE Computer Society Press, May 1990. (p 54)

[CR93] C. Chen and N. Roussopoulos. Adaptive database buffer allocation using query feedback. In *Proc. of the 19th VLDB Conference, Dublin, Ireland*, 1993. (p 37)

[CR96] Andrew Choi and Manfred Ruschitzka. Optimal management of dynamic buffer caches. *Performance Evaluation*, 26:239–262, 1996. (p 39)

[CY89] D. W. Cornell and P. S. Yu. Integration of buffer management and query optimization in relational database environment. In *Proc. of the 15th Int. Conf. on Very Large Data Bases*, pages 247–255, Amsterdam, August 1989. (p 36)

[Den68] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968. (p 36)

[DWAP94] M. Dahlin, R. Wang, T. Anderson, and D. 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, Nov 1994. (p 115)

[EGCD73] Jr. Edward G. Coffman and Peter J. Denning. *Operating Systems Theory*. Prentice Hall, 1973. (p 39)

[FKM+91] A. Fiat, R.M. Karp, M. Luby L. A. McGeoch, D.D. Sleator, and N.E. Young. Competitive paging algorithms. *Journal of Algorithms*, 12:685–699, 1991. (pp 145, 157, 159, 169, 172, 179)

[FNS91] Christos Faloutsos, Raymond Ng, and Timos Sellis. Predictive load control for flexibile buffer allocation. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 265–274, 1991. (p 37)

[FO71] R. J. Feiertag and E. I. Organisk. The Multics Input/Output system. In *Proc. of the 3rd Symp. on Operating System Principles*, pages 35–41, 1971. (p 34)

[FR94] Amos Fiat and Moty Ricklin. Competitive algorithms for the weighted server problem. *Theoretical Computer Science*, 130:85–99, 1994. (p 147)

[GA93] James Griffioen and Randy Appleton. Automatic prefetching in a WAN. In *Proc. of the IEEE Workshop on Advances in Parallel and Distributed Systems*, pages 8–12, October 1993. (p 34)

[GA94] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. In *USENIX Summer 1994 Technical Conference*, pages 197–208, June, 1994. (p 34)

[GA95] J. Griffioen and R. Appleton. Performance measurements of automatic prefetching. In *Proc. of the ISCA International Conference on Parallel and Distributed Computing Systems*, September 1995. (p 34)

[GA96] J. Griffioen and R. Appleton. The design, implementation, and evaluation of a predictive caching file system. Technical Report CS-264-96, Kentucky University, June 1996. (p 35)

[GJ91] A.S. Grimshaw and E.C. Loyot Jr. ELFS: Object-oriented extensible file systems. Technical Report Computer Science Technical Report No. TR-91-14, University of Virginia, 1991. (p 34)

[HC92] Kieran Harty and David R. Cheriton. Application-controlled physical memory using external page-cache management. In *The Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 187–197, October 1992. (p 38)

[Hol94] Mark Holland. *On-Line Data Reconstruction in Redundant Disk Arrays*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1994. (pp 54, 56)

[HP96] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 1996. (p 115)

[IS95] S. Irani and S. Seiden. Randomized algorithms for metrical task systems. In *Workshop on Algorithms and Data Structures*, 1995. (p 147)

[KE90] David Kotz and Carla Schlatter Ellis. Prefetching in file systems for MIMD multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):218–230, April 1990. (p 37)

[KE91] David Kotz and Carla Schlatter Ellis. Practical prefetching techniques for parallel file systems. In *First Intl. Conf. on Parallel and Distributed Information Systems*, pages 182–189, Miami Beach, Florida, December 4–6 1991. (p 37)

[KE93] David Kotz and Carla Schlatter Ellis. Practical prefetching techniques for multiprocessor file systems. *Journal of Distributed and Parallel Databases*, 1(1):33–51, January, 1993. (p 37)

[KK96a] T. Kimbrel and A. Karlin. Integrated parallel prefetching and caching. Technical Report UW-CSE-96-01-10, University of Washington, 1996. (p 31)

[KK96b] T. Kimbrel and A. Karlin. Near-optimal parallel prefetching and caching. In *Proceedings of the 1996 IEEE Symposium on Foundations of Computer Science*, October, 1996. (pp 64, 181)

[KLVA93] Keith Krueger, David Loftesness, Amin Vahdat, and Tom Anderson. Tools for the development of application-specific virtual memory management. In *OOPSLA 1993 Conference Proceedings*, pages 48–64, October 1993. (p 38)

[KMMO94] A.R. Karlin, M.S. Manasse, L.A. McGeoch, and S. Owicki. Competitive randomized algorithms for nonuniform problems. *Algorithmica*, 11:542–571, 1994. (pp 146, 151, 172)

[KMRS88] A.R. Karlin, M.S. Manasse, L. Rudolph, and D.D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1):79–119, 1988. (p 138)

[Kor90] Kim Korner. Intelligent caching for remote file service. In *Proceedings of the 10th Intl. Conf. on Distributed Computing Systems*, pages 220–226, 1990. (p 35)

[Kot94] D. Kotz. Disk-directed I/O for MIMD multiprocessors. In *Proc. of the 1st USENIX Symp. on Operating Systems Design and Implementation, Monterey, CA*, pages 61–74, November 1994. (p 37)

[KPR94] G. Kuenning, G. Popek, and P. Reiher. An analysis of trace data for predictive file caching in mobile computing. In *Proc. of the Summer 1994 USENIX conference*, June, 1994. (p 35)

[KRR91] H.J. Karloff, Y. Rabani, and Y. Ravid. Lower bounds for randomized $k$-server algorithms. In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing*, 1991. (pp 143, 146, 147, 149, 150)

[KS92] James J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 6(1):1–25, February 1992. (p 34)

[KTP+96] T. Kimbrel, A. Tomkins, R.H. Patterson, B. Bershad, P. Cao, E.W. Felten, G. Gibson, A. Karlin, and K. Li. A trace-driven comparison of algorithms for parallel prefetching and caching. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, pages 19–34, 1996. (pp 20, 30, 33, 36, 59, 66, 70, 87, 132, 133)

[KTR94] David Kotz, Song Bac Toh, and Sriram Radhakrishnan. A detailed simulation model of the hp 97650 disk drive. Technical Report PCS-TR94-220, Dartmouth University, July 18, 1994. (pp 42, 56)

[LD97] Hui Lei and Dan Duchamp. An analytical approach to file prefetching. In *1997 USENIX Annual Technical Conference*, Anaheim CA, January 1997. (p 35)

[Lee89] Edward K. Lee. The performance of parity placements in disk arrays. *IEEE Transactions on Computers*, 42(6):651–664, June 1989. (pp 32, 54, 56)

[LHR90] Kai-Fu Lee, Hsiao-Wuen Hon, and Raj Reddy. An overview of the SPHINX speech recognition system. *IEEE Transactions on Acoustics, Speech and Signal Processing, (USA)*, 38(1):35–45, Jan, 1990. (p 46)

[LK91] Edward K. Lee and Randy H. Katz. Performance consequences of parity placement in disk arrays. In *ASPLOS4*, pages 190–199. ACM, 1991. (p 54)

[MA90] Dylan McNamee and Katherine Armstrong. Extending the Mach external pager interface to accomodate user-level page replacement policies. In *Proc. of the USENIX Association Mach Workshop*, pages 17–29, 1990. (p 38)

[MDK96] T. Mowry, A. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation*, 1996. (pp 18, 29, 34)

[MJLF84] M. K. McKusick, W. J. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for Unix. *ACM Trans. on Computer Systems*, 2(3):181–197, August 1984. (p 34)

[MK91] L. W. McVoy and S. R. Kleiman. Extent-like performance from a UNIX file system. In *Proc. of 1991 Winter USENIX*, pages 33–43, 1991. (p 34)

[MLG92] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *The Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992. (p 38)

[MMS88a] M.S. Manasse, L.A. McGeoch, and D.D. Sleator. Competitive algorithms for on-line problems. In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing*, pages 322–333, 1988. (pp 22, 146)

[MMS88b] M.S. Manasse, L.A. McGeoch, and D.D. Sleator. Competitive algorithms for server problems. Technical Report CMU-CS-88-197, Carnegie Mellon University, 1988. (pp 140, 145, 146)

[MMS90] M.S. Manasse, L.A. McGeoch, and D.D. Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11:208–230, 1990. (pp 140, 166, 170, 172)

[Nat89] National Center for Supercomputing Applications. XDataSlice for the X window system. Technical Report http://www.nsca.uiuc.edu/, University of Illinois at Urbana-Champaign, 1989. (p 44)

[NFS91] Raymond Ng, Christos Faloutsos, and Timos Sellis. Flexible buffer allocation based on marginal gains. In *Proc. of the 1991 ACM Conf. on Management of Data (SIGMOD)*, pages 387–396, 1991. (pp 37, 60)

[OOW93] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proc. of the 1993 ACM SIGMOD Conference on Management of Data*, pages 297–306, May 1993. (p 36)

[PF76] B.G. Prieve and R.S. Fabry. VMIN — an optimal variable-space page replacement algorithm. *Communications of the ACM*, 19(5):295–297, 1976. (p 39)

[PG94] R. Hugo Patterson and Garth Gibson. Exposing I/O concurrency with informed prefetching. In *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*, pages 7–16, September 1994. Unpublished version in lab. (pp 18, 45)

[PGG+95] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. Informed prefetching and caching. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 79–95, December, 1995. (pp 18, 33, 36, 44, 59, 60, 80)

[PGK88] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *International Conference on Management of Data (SIGMOD)*, pages 109–116, June 1988. (p 18)

[PK94] Christos Papadimitriou and Elias Koutsoupias. The work function algorithm is competitive. In STOC 94, pages 507–511, May, 1994. (pp 146, 166)

[PZ91] Mark Palmer and Stanley B. Zdonik. FIDO: A cache that learns to fetch. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 255–264, September, 1991. (p 35)

[RL92] Anne Rogers and Kai Li. Software support for speculative loads. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, October 1992. (p 38)

[RS89] P. Raghavan and M. Snir. Memory versus randomization in online algorithms. In *Proc. ICALP*, 1989. (p 146)

[RW94] Chris Ruemmler and John Wilkes. An introduction to disk drive modelling. *IEEE Computer*, 27(3):17–28, March, 1994. (pp 42, 54, 56)

[SAC+79] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. of the 1979 ACM SIGMOD*, pages 23–34, Boston, MA, 1979. (p 35)

[SCO90] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited. In *Proc. of USENIX Winter 1990 Technical Conference*, pages 313–324, 1990. (pp 42, 56)

[SF94] A. Stathopoulos and C. F. Fischer. A Davidson program for finding a few selected extreme eigenpairs of a large, sparse, real, symmetric matrix. *Computer Physics Communications*, 79:268–290, 1994. (p 44)

[SGM86] K. Salem and H. Garcia-Molina. Disk striping. In *Proc. of the 2nd IEEE Int. Conf. on Data Engineering*, 1986. (p 18)

[Smi78] Alan J. Smith. Sequential program prefetching in memory hierarchies. *IEEE Computer*, 11(12):7–21, December 1978. (p 38)

[Smi85] A.J. Smith. Disk cache — miss ratio analysis and design considerations. *ACM Trans. on Computer Systems*, 3(3):161–203, August 1985. (pp 18, 34)

[SR86] M. Stonebraker and L.A. Rowe. The design of POSTGRES. In *Proceedings of the ACM SIGMOD 1986 International Conference on Management of Data, Washington, DC*, pages 28–30, 1986. (p 45)

[SRH90] M. Stonebraker, L.A. Rowe, and M. Horohama. The implementation of POSTGRES. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):125–142, March, 1990. (p 45)

[SS82] G. M. Sacco and M. Schkolnick. A mechanism for managing the buffer pool in a relational database using the hot set model. In *Proc. of the 8th Int. Conf. on Very Large Data Bases*, pages 257–262, September 1982. (p 36)

[SS86] G. M. Sacco and M. Schkolnick. Buffer management in relational database systems. *ACM Transactions on Database Systems*, 11(4):474–498, December 1986. (p 36)

[SS95] D. Steere and M. Satyanarayanan. Using Dynamic Sets to overcome high I/O latencies during search. In *Proc. of the 5th Workshop on Hot Topics in Operating Systems, Orcas Island, WA*, pages 136–140, May 4–5, 1995. (p 33)

[ST85] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985. (pp 138, 145)

[Sto81] Michael Stonebraker. Operating system support for database management. *Communications of the ACM*, 24(7):412–418, July 1981. (p 35)

[TD91] C. Tait and D. Duchamp. Detection and exploitation of file working sets. In *Proc. Eleventh Intl. Conf. on Distributed Computing Systems*, pages 2–9, IEEE, 1991. (p 35)

[TE93] Dean M. Tullsen and Susan J. Eggers. Limitations of cache prefetching on a bus-based multiprocessor. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 278–288, May 1993. (p 38)

[TPG97] A. Tomkins, R.H. Patterson, and G. Gibson. Informed multi-process prefetching and caching. In *Proceedings of the ACM SIGMETRICS*, pages 100–114, 1997. (pp 20, 36, 101)

[Tri79] K.S. Trivedi. An analysis of prepaging. *Computing*, 22:191–210, 1979. (p 38)

[WLAG93] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proc. of the 1993 ACM SIGOPS*, pages 203–216, 1993. (p 29)

[WM92] S. Wu and U. Manber. AGREP — a fast approximate pattern-matching tool. In *Proc. of the 1992 Winter USENIX Conference, San Francisco, CA*, pages 20–24, Jan, 1992. (p 47)

[YC91] Philip S. Yu and Douglas W. Cornell. Optimal buffer allocation in a multi-query environment, 1991. (p 36)

[You91] Neal E. Young. On-line caching as cache size varies. In *Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 241–250, 1991. (p 146)

# Index