USING SYSTEM STRUCTURE AND SEMANTICS FOR VALIDATING AND OPTIMIZING PERFORMANCE OF MULTI-TIER STORAGE SYSTEMS

by

Gokul Soundararajan

A thesis submitted in conformity with the requirements
for the degree of Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto

Copyright © 2010 by Gokul Soundararajan
Abstract

Using System Structure and Semantics for Validating and Optimizing Performance of Multi-tier Storage Systems

Gokul Soundararajan
Doctor of Philosophy
Graduate Department of Electrical and Computer Engineering
University of Toronto
2010

Modern persistent storage systems must balance two competing imperatives: they must meet strict application-level performance goals and they must reduce the operating costs. The current techniques of either manual tuning by administrators or by over-provisioning resources are either time-consuming or expensive. Therefore, to reduce the costs of management, automated performance-tuning solutions are needed.

To address this need, we develop and evaluate algorithms centered around the key thesis that a holistic semantic-aware view of the application and system is needed for automatically tuning and validating the performance of multi-tier storage systems. We obtain this global system view by leveraging structural and semantic information available at each tier and by making this information available to all tiers. Specifically, we develop two key building blocks: (i) context-awareness, where information about the application structure and semantics is exchanged between the tiers, and (ii) dynamic performance models that use the structure of the system to build lightweight resource-to-performance mappings quickly.

We implement a prototype storage system, called Akash, based on commodity components. This prototype enables us to study all above scenarios in a realistic rendering of a modern multi-tier storage system. We also develop a runtime tool, Dena, to analyze the performance and behavior of multi-tier server systems.

We apply these tools and techniques in three real-world scenarios. First, we leverage application context-awareness at the storage server in order to improve the performance of I/O prefetching. Tracking application access patterns per context enables us to improve
the prediction accuracy for future access patterns, over existing algorithms, where the high interleaving of I/O accesses from different contexts make access patterns hard to recognize. Second, we build and leverage dynamic performance models for resource allocation, providing consistent and predictable performance, corresponding to pre-determined application goals. We show that our dynamic resource allocation algorithms minimize the interference effects between e-commerce applications sharing a common infrastructure. Third, we introduce a high-level paradigm for interactively validating system performance by the system administrator. The administrator leverages existing performance models and other semantic knowledge about the system in order to discover bottlenecks and other opportunities for performance improvements. Our evaluation shows that our techniques enable significant improvements in performance over current approaches.
I do not know what I may appear to the world, but to myself I seem to have been only like a boy playing on the sea-shore, and diverting myself in now and then finding a smoother pebble or a prettier shell than ordinary, whilst the great ocean of truth lay all undiscovered before me. — Isaac Newton
Acknowledgements

Who else but Isaac Newton can best sum up the experiences of the graduate student? Indeed, over the past few years, I have been a boy playing on the sea-shore attempting to fathom computer science by studying its contents – that is, I have tried to understand the sea by studying the pebbles and shells that reach the sea-shore. I am fortunate for being given the opportunity to explore the intricacies of modern computer systems and truly grateful for given the time to understand them. I acknowledge and thank the many people who have helped me along this journey.

First, I would like to thank my family without whose love and support this Ph.D. would have been impossible. My parents prepared me well by providing the freedom to explore my curiosities. They have always encouraged me to satisfy my desire to learn and understand how things work; a stream of broken toys can attest to that! I especially thank my sister, Subhashini, for keeping me in high spirits. During our shared daily commute to Toronto over the last year, she has given me feedback on my presentation and slides. Even now, the lively discussions at the kitchen table allow me to organize my rambling thoughts and develop a cohesive idea.

I would also like to thank my advisor, Cristiana Amza, for her support and guidance over the past seven years. She gave me the freedom to explore a new topic (storage systems) and was patient with me as I built the Akash platform. Her mentorship has allowed me to hone my skills and has given me the confidence to finally graduate and take the plunge into the ocean of truth. I thank Angela Demke Brown, Ashvin Goel, Ken Salem, and Michael Stumm for serving on my thesis committee. Ashvin helped me understand systems with his hands-on help with the design of database provisioning system during my Masters thesis. I admire Michael’s insights and his ability to manage time (and multiple timezones). Finally, I want to acknowledge the extraordinary effort of the thesis committee and Professor Andreas Moshovos to attend my defense by teleconference despite the havoc caused by the eruption of Eyjafjallajökull. I appreciate your effort.

My internships at Amazon and Microsoft Research have allowed me to gain a different perspective. I thank Mikhail Graber, Grant McAlister, Rajesh Sheth, and Swaminathan Sivasubramanian for making the Amazon internship a great learning experience. I thank Mahesh Balakrishnan and Vijayan Prabhakaran for making my Microsoft internship experience fun and exciting. Finally, I thank my colleagues: Ekin Akkus, Reza Azimi, Mihai Burcea, Jin Chen, Saeed Ghanbari, Daniel Lupei, Madalin Mihailescu, Don Pinto, Adrian Popescu, Livio Soares, David Tam, and Weihan Wang for their support and help.
# Contents

1 Introduction ................................................. 1
   1.1 Contributions ........................................ 7
   1.2 Outline ................................................ 8

2 Background .................................................. 9
   2.1 The Emergence of Modern Datacenters ................. 9
   2.2 Design of Multi-tier Server Systems .................. 11
   2.3 Optimizing Application Performance ................... 11
   2.4 Dynamic Resource Allocation .......................... 14
   2.5 Validating System Performance ........................ 18

3 Optimizing Application Performance ....................... 20
   3.1 Introduction ........................................... 20
   3.2 Motivation .............................................. 23
   3.3 Context-aware Mining and Prefetching ................. 26
      3.3.1 Overview ........................................... 26
      3.3.2 Tracking High-Level Contexts ..................... 27
      3.3.3 Determining Blocks to Prefetch ................... 28
      3.3.4 Issuing Prefetch Requests ........................ 29
   3.4 Example of Context-Aware Prefetching ................. 30
   3.5 Prototype Implementation .............................. 33
      3.5.1 Interfaces and Communication ...................... 34
      3.5.2 Modules of the Akash Storage Server .............. 34
      3.5.3 Changes made to Existing Code .................... 37
   3.6 Evaluation .............................................. 38
      3.6.1 Prefetching Algorithms used for Comparison ...... 38
      3.6.2 Benchmarks ......................................... 40
      3.6.3 Methodology ....................................... 41
   3.7 Results ................................................ 42
      3.7.1 TPC-W E-Commerce Workload ........................ 42
      3.7.2 RUBiS Auctions Workload ............................ 44
      3.7.3 DBT-2 Transaction Processing Workload ............ 44
   3.8 Detailed Analysis ...................................... 46
      3.8.1 Detailed Comparison of Prefetching Effectiveness . 46
      3.8.2 Benefit of Context Awareness ...................... 47
4 Dynamic Resource Allocation

4.1 Introduction .................................................. 52
4.2 Motivation ..................................................... 55
  4.2.1 Sharing Storage Bandwidth within an Operating System .... 56
  4.2.2 Sharing Cache Space and Disk Bandwidth in a Storage Server ... 57
4.3 Problem Statement and Overview of Solution ..................... 60
  4.3.1 Problem Statement ........................................ 61
  4.3.2 Overview of Solution ...................................... 61
4.4 Building Approximate Performance Models ......................... 62
  4.4.1 Key Assumptions and Ideas ................................ 62
  4.4.2 Modeling the Performance of a Two-Level Cache Hierarchy .... 63
  4.4.3 Modeling the Disk Performance ............................ 65
  4.4.4 Computing the Overall Performance Model .................. 66
4.5 Runtime Refinement of Performance Models ....................... 67
  4.5.1 Limitations of the Cache Model ........................... 68
  4.5.2 Limitations of the Disk Model ............................ 71
  4.5.3 Iterative Refinement of Performance Models ................. 71
4.6 Finding the Optimal Configuration ................................ 74
4.7 Supporting General Utility Functions ................................ 74
4.8 Analytical Derivation of a Sample Scenario ....................... 75
4.9 Prototype Implementation ...................................... 78
4.10 Evaluation .................................................. 78
  4.10.1 Algorithms used for Comparison ......................... 79
  4.10.2 Benchmarks ............................................. 79
  4.10.3 Evaluation Methodology .................................. 80
  4.10.4 Sampling Methodology ................................... 81
4.11 Results .................................................. 82
  4.11.1 Miss-Ratio Curves ...................................... 82
  4.11.2 Running Two Applications ............................... 82
  4.11.3 Running Four Applications .............................. 93
  4.11.4 Resource Allocation with Utility Functions ............... 94
4.12 Detailed Analysis ........................................... 95
  4.12.1 Accuracy of the Two-Level Cache Model ................. 96
  4.12.2 Accuracy of the Quanta-based Scheduler Approximation ..... 98
  4.12.3 Model Refinement with Runtime Sampling ................. 98
  4.12.4 Cost of Obtaining Experimental Samples .................. 99
4.13 Summary .................................................. 101
Chapter 1

Introduction

In this dissertation, we design, implement, and experimentally evaluate novel techniques for predicting, optimizing and validating the performance of multi-tier persistence systems managing data in modern data centers [11].

Modern data centers consist of multiple software layers, including web/application server front-ends, database servers running on top of the operating system, and storage servers at the lowest level, as shown in Figure 1.1. Database systems and storage servers are collectively referred to as persistence systems by cutting edge service providers, such as, Amazon.com [27]. In order to reduce operational costs, service providers schedule several database applications to run on the same persistence system, thus multiplexing resource usage over a set of commodity components. Operating systems (OSes) have traditionally supported the sharing of resources by several concurrently running applications albeit oblivious to application-level goals, such as by optimizing for disk seeks when issuing I/O requests, for example [91]. Unfortunately, while optimizing resource usage, uncontrolled sharing can lead to application-level performance degradation, due to thrashing or starvation. Limited mechanisms for implementing fairness and priorities do exist in some versions of modern OSes, e.g., in Linux [79, 105] regardless, severe interference for resources, such as, the CPU, memory, and disk bandwidth can still occur.

Uncontrolled resource sharing thus creates a problem for service providers, because respecting application-level requirements is considered paramount in modern environments. For example, in recent years, stricter customer contracts and the importance of customer satisfaction in a highly competitive market, have advanced application-level goals, such as, average latency bounds or throughput guarantees to first class citizen status [27]. As a result, in state-of-the-art environments, such as Amazon.com, much effort is expended on
Figure 1.1: **Multi-tier Server Systems:** We show the different tiers in modern datacenters. It consists of front-end web servers, database servers, and storage servers.
finessing the application performance versus resource usage trade-off, and investigating performance bottlenecks, usually in a manual, time-consuming fashion. A variety of low-level profiling tools to help highly skilled administrators exist for this purpose but, in the common case, the administrator still needs to compile, and fine-tune a variety of metrics across the system, by poring over system logs, or low-level statistics, which is error-prone, expensive in terms of the required level of skill, nerve wracking, time consuming, or all four. The alternative solution is to sacrifice resource usage, by conservatively over-provisioning resources to accommodate overall expected peak usage for all applications. While new hardware-level power-saving mechanisms are increasingly being deployed to reduce energy consumption and cooling requirements of idle or partially used servers, neither manual fine-tuning, nor resource over-provisioning is a satisfactory solution in terms of cost-effectiveness.

To address the acute need for a cost-effective performance-tuning solution, in this dissertation, we focus on automating the process of finding an approximation of the optimal resource allocation configuration for providing a pre-determined application-level performance guarantee. We further introduce automatic application-aware performance optimization techniques for a given set of resources. Finally, we study high-level paradigms and advanced tools for helping the administrator validate system performance.

Our central claim is that:

*Semantic information, such as, application contexts, system structure and functionality principles, is crucial towards designing feasible solutions for dynamic resource allocation, improved and predictable performance, and easier performance validation.*

Intuitively, the underlying reason justifying a semantically-driven approach is the separation of concerns employed in modern commodity-based multi-tier persistence systems, which leads to a *semantic gap* between the application and different levels of the system. The lower layers lack higher-level knowledge about the application. Conversely, lower-level knowledge e.g., data layout, is lacking at the higher levels. An aggravating factor, which widens the semantic gap, is the use of resource virtualization as a means of reducing management costs [10]. All such layers of indirection lead to reduced cohesion between the application and the underlying hardware.

We argue that, unless semantic information is explicitly known, and leveraged throughout the system in a holistic manner, application-level goals by necessity would be ignored at the lower levels of the system, application-centric performance optimization opportunities
would be missed, and understanding application and system performance can be nothing but challenging. Finally, semantic information is important in guiding the search towards a solution, for time consuming processes, such as optimal resource allocation, and system validation [82].

Our key ideas for successfully bridging this semantic gap in achieving our goals are: (i) to design minimally intrusive techniques for tracking application contexts within pre-existing commodity multi-tier storage systems and (ii) to leverage application contexts within lightweight, dynamically refinable performance models for applications.

We define tracking application contexts as the ability to recognize: (i) the application’s identity for resource accesses at any level within the system, (ii) the application’s structural semantics and goals, such as, the nested scopes of application interactions and queries within interactions, and their associated performance guarantees at any level within the system, and (iii) the set of environmental factors that may affect application performance, such as, the structure of the cache hierarchy and the cache replacement policies in effect.

We define a performance model as a mathematical function calculating the application performance at a given resource configuration, or as an expression of the system structure in varying detail. For example, a simple model may be directly specified to the system as a high-level function that expresses generic laws governing system behavior, e.g., Little’s law [46], expressing the inverse relationship between throughput and response time in a closed-loop system. At the other end of the spectrum, a low-level model may provide a detailed estimate of the performance of an application, given different amounts of resources, e.g., the amount of memory and the disk bandwidth fraction. Regardless of the level of detail, performance models provide an insight into the overall behavior of the system.

Our approach is to use known application contexts in order to design and develop fast and adaptive model building algorithms, with dynamic refinement towards optimal performance solutions. We assume that overall performance goals are defined to the system in some way. These goals can be as varied as: desired per-application latencies, utilities, or priorities, overall application throughput, aggregate latency requirements, and so on. We leverage system semantics in order to build a performance model skeleton based on well known system structure and functionality principles, such as, known data flow and API’s between tiers, cache replacement algorithms and known laws of closed loop systems, properties of cache miss rates, etc. Based on this skeleton embodiment of high level system knowledge, our performance models are dynamically refined on-line, based on experimentally sampled application and system metrics.
We implement the above building blocks in our own cluster-based storage prototype, called Akash. We evaluate and showcase our powerful algorithms and approaches leveraging these building blocks in the following three real-world application scenarios:

1. **Optimizing Application Performance: Leveraging Context Awareness for Improving Prefetching Performance**

   We propose and evaluate QuickMine, a novel, lightweight and minimally intrusive method for context aware prefetching. Using QuickMine, we capture application contexts, such as, a transaction or query, and we leverage them for context-aware prediction, and improved prefetching effectiveness in the storage cache.

   Our Akash-based prototype shows that context-aware prefetching clearly out-performs existing context-oblivious prefetching algorithms, resulting in factors of up to 2 improvements in application latency for two e-commerce workloads with repeatable access patterns, TPC-W and RUBiS.


   We introduce a novel multi-tier resource allocator to dynamically allocate resources for multiple database servers running on shared cluster-based storage. Our multi-tier resource allocation involves proportioning the database and storage server caches, and the storage bandwidth between applications according to overall performance goals.

   We use a combination of dynamic performance models and on-line sampling to arrive at near-optimal configurations within minutes. The key idea is to incorporate access tracking and known resource dependencies e.g., due to cache replacement policies, into our performance model. We implement our dynamic resource allocation algorithm within our storage system prototype, Akash. In our experimental evaluation, we use both micro-benchmarks and the industry standard benchmarks TPC-W and TPC-C. Our results show that multi-resource partitioning allows for performance improvements of up to a factor of 6 for synthetic benchmarks, and a factor of 4 for industry-standard benchmarks compared to state-of-the-art single-resource controllers, and their ad-hoc combination. At the same time, the results show that our techniques achieve performance that is within 20% of the performance achieved by an exhaustive approach, but in a fraction of the time.

We study techniques and tools that allow the administrator to gain more insight into multi-tier storage systems, thus enabling performance validation, and administrator-directed performance diagnosis. For this purpose, we leverage our insights into performance models, and a novel declarative language, called SelfTalk, that allows administrators and users to query and understand the status of a large scale system. SelfTalk is sufficiently expressive to encode administrator high level hypotheses/expectations about system semantics, and normal system behavior elicited from known performance models [32, 33]. SelfTalk works in conjunction with Dena, a runtime support system designed to help system administrators validate their hypotheses about system behavior and diagnose system performance interactively.

Given a user’s hypothesis, Dena instantiates and validates it using actual monitored data within specific system contexts. We evaluate SelfTalk/Dena by posing several hypotheses about system behavior, and querying Dena to diagnose system performance anomalies in a multi-tier dynamic content server based on Akash. We find that Dena automatically validates or diagnoses a performance anomaly given an administrator’s expectation within 2 seconds, on over 50GB of monitoring data. Specifically, we validate the behavior of caches, request schedulers, and the interplay between the different tiers.
1.1 Contributions

This dissertation provides automatic tools and techniques to address the problem of application-centric (i) performance optimization (ii) resource allocation that satisfies pre-determined application requirements and (iii) performance validation in multi-tier storage systems. Specifically, we apply application context-awareness and dynamically built performance models in the following high impact realistic scenarios.

- We leverage application context-awareness at the storage server in order to improve the performance of I/O prefetching. Tracking application access patterns per context enables us to improve the prediction accuracy for future access patterns, over state-of-the-art algorithms, where the high interleaving of I/O accesses from different contexts make access patterns hard to recognize.

- We build and leverage dynamic performance models for resource allocation, providing consistent and predictable performance, corresponding to pre-determined application goals. We show that our dynamic resource allocation algorithms minimize the interference effects between e-commerce applications sharing a common storage infrastructure.

- We introduce a high-level paradigm for interactively validating system performance by the system administrator. The administrator leverages pre-existing performance models and other semantic knowledge about the system in order to discover bottlenecks and other opportunities for performance improvements.

We implement a prototype of a virtual storage system, called Akash, based on commodity components. This prototype enables us to study all above scenarios in a realistic rendering of a modern multi-tier server system. We also develop a declarative language, SelfTalk and a runtime tool, Dena, to analyze the performance of multi-tier server systems, and the interactions between its many components. Finally, we acknowledge that parts of this dissertation have been published in prior publications [32, 33, 95, 97, 98, 99].
1.2 Outline

The outline of this dissertation is as follows. Chapter 2 provides the necessary background on the design of datacenters and discusses previous work tackling the issues presented in the subsequent chapters. Chapter 3 describes the context-aware prefetching technique. Chapter 4 describes the dynamic resource allocation algorithm to improve the overall performance of the multi-tier storage systems. Chapter 5 presents our declarative language and runtime that allows administrators to understand the behavior and diagnose misbehavior of multi-tier storage systems. Chapter 6 presents related work. Chapter 7 concludes the dissertation and outlines avenues for future work.
Chapter 2

Background

In this chapter, we provide the context for the techniques presented in this dissertation. We begin by describing the high-level design of modern datacenters and focus on the performance bottlenecks that arise from the commodification and consolidation of storage resources. We also discuss previous work in addressing these challenges and introduce the techniques we elaborate on in the subsequent chapters.

2.1 The Emergence of Modern Datacenters

Early Internet applications were conceptually simple – displaying static webpages, while modern Internet services are more complex. They provide a wide variety of services from web-based equivalents of many desktop-based applications such as word processing and photo editing, to web-based commerce such as online shopping and banking, and to web-based entertainment such as digital content downloads and live media streaming. The paradigm shift to web-based computing has led to the emergence and the subsequent growth of modern datacenters – where most Internet companies today use the datacenter model to reduce both (i) software costs, and (ii) hardware and operational costs [11].

The reduction in software costs is clear. Internet services follow the client-server computing model, where the client (a user accessing through a web browser) interacts with a service (the software runs on the server managed by the company). This computing model allows the Internet company to produce one version of the software – that is, a version that can run on its server hardware. This frees the company from testing its software on a myriad of different system configurations, to different hardware configurations, and to the different operating systems that may be running the users’ desktops. A secondary benefit
of this shift to server-side computing is that it allows for faster software development and upgrades. Since the software is installed on the servers of the Internet company, changes to the software due to either bug fixes or new features can be quickly deployed.

The reduction in hardware and operational costs is not so clear. On one hand, the Internet company achieves economies of scale by sharing its servers among thousands of users [11]. On the other hand, the Internet company must also provide good user experience. The user of an online service expects the same responsiveness from the web-based service (e.g., web-based word processing) that she expects from a desktop application. With these competing requirements, the Internet company not only needs to use many servers to provide good user experience but also needs to consolidate resources to reduce costs. The right balance is difficult to obtain and poses an interesting research challenge – “How does one consolidate resources to maximize the performance of the physical hardware while maintaining good user experience?” We address this question in this dissertation. We focus on persistent storage systems as they are an integral part of the server software stack and improving its performance helps in reducing the overall costs for the Internet company.

Storage systems are software systems that provide the storage and retrieval of data. They range from database systems and file systems at the high-level to storage servers and disks at the low-level. We focus on improving the performance of storage systems for two reasons. First, in many current applications, the server-side processing consists of only software to manage users' data reliably and with good performance. For example, consider the web-based services: web-based word processing and digital media streaming; in both services, the primary processing of data occurs on the user’s desktop i.e., rendering the document or video on the user’s screen and the server-side processing consists of applying the user’s changes to the document or delivering the next frame in the video. In fact, Amazon.com reports that the performance of storage systems directly corresponds to the performance of its e-commerce site [27]. Second, the volume of digital data stored at the servers is increasing. For example, in 2008, Amazon S3\(^1\) reported that it stores over 40 billion objects and the number of objects is roughly doubling every year [41]. The combination of the above two trends indicate that storage services account for a large fraction of the computing budget and methods to improve the performance of storage systems will lead to significant overall cost savings.

\(^1\)Amazon Simple Storage Service. [http://aws.amazon.com](http://aws.amazon.com)
2.2 Design of Multi-tier Server Systems

Modern enterprise e-commerce systems consist of multiple software layers including web/application server front-ends, database servers running on top of the operating system, and storage servers at the lowest level. Figure 1.1 illustrates this design. In addition, each of these software layers is shared among several applications (i.e., server consolidation) to reduce hardware and management costs.

A client request for dynamic content causes the web server to invoke a function in the application server. The application server executes the application program, issues SQL queries, one at a time, to the database which in turn fetches data from the storage server. The results from the database server are sent to the the application server that formats the results as an HTML page. The web server then returns this page in an HTTP response to the client. Within this multi-tier dynamic content system, we focus on the persistent storage tiers, e.g., the database and storage servers; specifically, our persistent storage system is built using MySQL (as the database server) and Akash (as the storage server).

In the following sections, we discuss three areas for improvement in a multi-tier persistent storage system: (i) optimizing application performance, (ii) dynamically allocating resources and, (iii) diagnosing and validating performance.

2.3 Optimizing Application Performance

We need to use resources effectively in all tiers to improve application performance. In the case of persistent storage systems, accessing the disk is the major bottleneck [96]. Previous work has shown that caching and prefetching of data can improve the performance of fetching data from storage systems [48, 50, 60, 64, 73, 111]. Data caching is a technique where the locality of access is exploited by keeping some of the data in memory to avoid repeatedly fetching the data from disk. In more detail, applications tend to exhibit temporal locality – where recently fetched data is re-accessed within a short time interval. Caching exploits this pattern by storing the recently accessed data in memory, thus avoiding the costly disk access. Prefetching, on the other hand, exploits the tendency of applications to access data in predictable patterns (e.g., sequentially) so future data accesses can be predicted based on past accesses. In this case, the data can be fetched in advance (i.e., prefetched) before the application requests the data item. Both caching and prefetching reduce the application’s perceived latency to access a data item as shown in previous work.
However, in newer multi-tier systems, the caches at both the database and the storage servers create a two-tier cache hierarchy and the effects of multi-tier caching have been less studied. The primary challenge to improving performance of multi-tier caches is the lack of information at the storage server either due to filtering of accesses by the database cache or due to the narrow interface between the database server and the storage server. The lack of information lowers the benefit of both caching and prefetching at the storage server.

The performance of caching is poor is due to lower temporal locality at the storage server where the accesses to the storage server cache are misses of the database cache – that is, a request to the storage server is made only if the item is not found in the database cache. In more detail, the database and storage caches typically employ the LRU (least-recently-used) cache replacement algorithm; this algorithm stores data for items that are repeatedly accessed in a short time interval. The I/O accesses arriving at the storage server are in turn for data items accessed over longer time intervals (i.e., have lower temporal localities). This runs counter to the heuristic used by the LRU replacement scheme leading to a lower caching benefit at the storage server. In addition, in a two-level cache hierarchy using the standard (uncoordinated) LRU replacement policy at both levels, any cache miss from the first-level cache results in bringing the needed block into the second-level cache of the cache hierarchy, before providing the requested block to first-level cache. It follows that the block is redundantly cached at all cache levels, which is called the inclusiveness property [111]. Therefore, if an application is given a certain cache quota at a first-level of the cache then, any cache quotas smaller than the first-level cache will be wasteful at the lower cache levels. The double caching of data items wastes valuable cache space when space is at a premium in server consolidation settings.

Prefetching algorithms perform poorly at the storage server because they lack high-level information of the application’s behavior needed to make informed decisions. The lack of information is primarily due to the narrow interface between the database server and the storage server. The interface between the database server and the storage server is a block I/O interface; the information passed from the database is the type of I/O (either read or write) to be performed, the offset in the block device to read/write, and the length of data accessed by the I/O request. As database servers are multi-threaded (typically each thread handles a different transaction), the stream of I/O accesses seen at the storage server is the interleaved stream of I/O accesses issued by each thread. The interleaving of I/Os coupled with the narrow interface between the database and storage server results in lower prefetching benefit. This occurs because the prefetching algorithm is unable to extract meaningful access
patterns from interleaved stream and lacks the high-level contextual information (e.g., thread identifiers) to reconstruct the original streams. In the following, we discuss how context-aware techniques improve the performance both caching and prefetching algorithms at the storage server.

The caching benefit at the storage server cache can be improved by designing cache replacement algorithms that are less sensitive to the lack of temporal locality [48, 50, 60, 64, 73, 111] or by enabling communication between caches to enable a coordinated cache replacement policy [60, 111, 112]. These cache replacement algorithms proposed for the storage server either use a different metric to make replacement decisions, e.g., recency as in [48], or augment the existing LRU replacement policy with access frequency to improve the caching benefit [50, 73]. These changes result in modest performance improvements of the storage cache [22]. Greater benefits can be obtained from the use of coordinated cache replacement algorithms such as the DEMOTE [111] scheme. The DEMOTE scheme sends block eviction hints or explicit demote operations from the client cache e.g., the database buffer pool, to the storage cache with the goal to maintain exclusiveness between the two caches [111]. When the client cache is about to evict a clean block, it sends the clean block to the storage cache using a special DEMOTE operation. The storage cache places the demoted block in its cache, ejecting another block if necessary. The storage cache also moves the blocks read by the client to the LRU (least-recently-used) position such they will be evicted before the demoted blocks. Similar coordination, using DBMS specific information, has been exploited by Li et al. [60], Liu et al. [64], and Yadgar et al. [112] for improving the coordination between the database and the storage server. The results from previous work has shown that these techniques can improve the caching benefit at the storage server.

However, caching algorithms can improve performance only if the application has a small memory footprint or exhibits locality in its accesses. If this is not the case, then the use of caches will not improve performance. Prefetching can be used to improve performance in these cases instead. The performance of prefetching algorithms at the storage server can be improved by using high-level contextual information. This is because the technique of prefetching data is inherently speculative – that is, prefetching algorithms analyze the past data access patterns and issue I/Os to fetch the data it believes will be accessed soon. Therefore, the predictions made by the prefetching algorithm can be improved, i.e., can be made more accurate, by utilizing additional information about the behavior of the application.

This information can be obtained indirectly where the application’s access patterns can be inferred based on application semantics [6, 7, 94] or explicitly where the application
provides hints about its future accesses [19, 34, 60, 77]. Inferring application behavior based on the application semantics [6, 7, 94] is a promising method; it allows the use of the application’s semantics, i.e., understanding the file system layout or a DBMS’s I/O access patterns, to predict future data accesses. However, the drawback of this approach is that the prediction logic needs to be rewritten if the application semantics is changed and it does not handle the case where the I/Os from different application threads are interleaved. A different method is to explicitly provide the storage system a list of future I/O accesses; this method provides a more efficient prefetching but requires an in-depth understanding of the DBMS internals [19, 60] and does not correct the interleaving of I/O accesses from multiple threads.

In contrast to existing techniques, which fall into the two extremes by either requiring explicit information on future accesses or by inferring application access behavior, we propose a novel and a minimally intrusive method for context-aware prefetching, called QuickMine, by adopting the positives of each approach: using application semantics by capturing the application’s behavior by mining past data access patterns and by using application hints by capturing application contexts, such as transaction or query, from tagging I/Os issued by the DBMS.

2.4 Dynamic Resource Allocation

The costs of management, power and cooling for large service providers hosting several applications are currently prohibitive, taking up more than 77% of the average company budget [102]. This is a major impediment to the efficiency of this industry, by limiting reinvestment, research and development. To achieve cost reductions, automated server consolidation techniques where several concurrent applications are multiplexed on each physical server are being designed and implemented in datacenters. However, determining and enforcing per-application resource quotas in the shared storage hierarchy, on the fly, poses a complex resource allocation problem spanning the database server and the storage server tiers. In its most general form, the problem of finding the optimal resource partitions leads to an NP-complete global resource allocation and control problem [82].

Let us consider the scenario where multiple applications are hosted in the consolidated storage environment as shown Figure 1.1. In this setup, the applications share the memory and disk resources that are controlled by different levels of the software stack, namely, the database server, the operating system, and the storage server. In many current systems, the
system exerts no control of resources (i.e., the lack of I/O priorities in the Linux operating system) or allocates resources to applications through different performance optimization loops, run independently at the different tiers (i.e., the CPU scheduler strives for fairness while the I/O scheduler strives to minimize seek times). Previous work on dynamic resource partitioning in shared server environments focused on partitioning a single resource within a single software tier at a time; specifically, previous work has studied methods to partition the CPU [10, 91, 105], partition the memory [13, 100, 104, 115], and partition the disk bandwidth [15, 17, 49, 65, 90, 103] in isolation.

The task of properly allocating the CPU has traditionally been the responsibility of the operating system (OS). Within the OS, many algorithms, such as round robin, priority, shortest job first (SJF), lottery scheduling, and multi-level feedback queues have been extensively studied and implemented in a variety of operating systems, such as Linux, BSD, and Microsoft Windows [91, 105]. These algorithms can be classified as using either a priority-based mechanism or a quanta-based mechanism. Under priority-based mechanisms, applications with low priority are prone to starvation. This makes such mechanisms inappropriate when the objective is to provide per application QoS guarantees. In contrast to priority-based mechanisms, quanta-based scheduling mechanisms guarantee that each application acquires a fair portion of the shared resource, e.g., as in lottery scheduling, where processes are assigned tickets proportional to their share [105]. More recently, virtual machines monitors (VMM), e.g., Xen, have implemented their CPU schedulers within the hypervisor to provide performance isolation for their virtual hosts [10]. Commercial DBMS products such the Oracle database server and Microsoft SQL Server implement CPU resource allocators within the database server by allowing the database administrator to specify limits CPU limits per (database) application [35, 71].

Dynamic memory partitioning between applications is typically performed using the miss-ratio curve (MRC) [115]. The MRC represents the page miss-ratio versus the memory size, and can be computed dynamically through Mattson’s Stack Algorithm [67]. The algorithm assigns memory increments iteratively to the application with the highest predicted miss-ratio benefit. MRC-based cache partitioning thus dynamically partitions the cache/memory to multiple applications, in a way to optimizes the aggregate miss-ratio. Dynamic memory allocation algorithms have also been studied in the VMWare ESX server [104] where the hypervisor estimates the working-set sizes of each VM and periodically adjusts each VM’s memory allocation such that performance goals are met. Adaptive cache management based on application patterns or query classes has been extensively studied in
database systems. Brown et al. [13] study schemes to ensure per-class response time goals in a system executing queries of multiple classes by sizing the different memory regions and, recently, IBM DB2 added the self-tuning memory manager (STMM) to size different memory regions [100]. The synergy between the database buffer pool and storage cache has been studied in multi-tier systems, but in the different context of cache replacement policies [60, 64, 111]. Coordinated cache replacement policies e.g., through block eviction/demote hints from the database server to the storage cache have been studied only in the context of improving the performance of a single application and not in the context of resource partitioning in the presence of multiple applications.

Several disk scheduling policies for enforcing disk bandwidth isolation between co-scheduled applications have been proposed [15, 17, 49, 65, 90, 103]. These existing algorithms work at the storage level and assume that the I/O deadlines, or disk bandwidth proportions are given a priori. For example, SLEDS [17], Façade [65], SFQ [49], and Argon [103] place a scheduling tier above the existing disk scheduler which controls the I/Os issued to the underlying disk. Argon [103] uses a quanta-based scheduler, while SLEDS [17] uses a leaky-bucket filter to throttle I/Os from clients exceeding their given fraction. Similarly, SFQ dynamically adjusts the deadline of I/Os to provide fair sharing of bandwidth. Finally, Cello [90] and YFQ [15] build QoS-aware disk schedulers, which make low-level scheduling decisions that strive to minimize seek times, as well as maintaining quality of service.

While methods to partition each resource has been studied in great detail, the previous approaches still fall short of providing effective resource partitioning due to the following two reasons. The first reason is that application QoS is usually expressed as a Service Level Objective (SLO) and, not as per-resource quotas; there is currently no automatic mechanism to accurately assign resource quotas for applications corresponding to a given application metric. The second reason that prevents these approaches from providing effective resource partitioning is the absence of coordination between different resource controllers located within different tiers. This absence of coordination might lead to situations where local partitioning optima do not lead to the global optimum; indeed local goals may conflict with each other, or with the per-application goals. For instance, the operating system may optimize for fairness in thread scheduling across applications, while the storage server may optimize for overall I/O performance (i.e., minimize disk latency). In other words, each resource controller optimizes local goals, oblivious to the goals of other resource controllers, and to the per-application SLO’s. There is little or no previous work on correlating priority or quota enforcement across several resources or software components [18, 75]. Specifically, conven-
tional wisdom for I/O prioritization holds that enforcing transaction priority at the CPU level automatically leads to the same priority enforcement at the storage level [1, 68]. We find that this assumption does not hold on state-of-the-art shared platforms. The resource allocation problem is further complicated when applications define different utilities (or priorities). For example, a high priority application i.e., the “gold customer” can experience substantial performance degradation due to I/O interference from co-scheduled applications, regardless of their respective CPU priorities.

Therefore, the main challenge in these modern enterprise environments is designing a strategy which adopts a holistic view of system resources; this strategy should efficiently allocate all resources to applications, and enforce per-application quotas in order to meet overall optimization goals e.g., overall application performance or service provider revenue. Towards controlling this interference we propose a dynamic global resource partitioning scheme that exploits the interdependency between the cache at the database server (i.e., the buffer pool) and the cache at the storage server, and between the caches and the underlying disk. We study methods for controlling interference among applications in this multi-tier storage hierarchy.

Our techniques are applicable to I/O intensive workloads where the working sets of applications together concurrently running together do not fit into the cache hierarchy. These situations are, and will remain common in the foreseeable future due to the following reasons. First, while both the buffer pool and storage server cache sizes are increasing, so do the memory demands of applications e.g., very large databases. Second, efficiently using the combined caching capabilities of database server and storage server is challenging even for a single application. Indeed, the potential for double caching of blocks, and the typically poor temporal locality of accesses that miss in the buffer pool lead to poor cache utilization in the storage level cache [22, 63]. Finally, running several applications on a cluster with consolidated storage, and/or on the same physical server exacerbates the above problems due to application interference for memory, hence the increased potential for capacity misses in the cache hierarchy.

Towards addressing the dynamic resource allocation problem in shared server environments, we introduce a novel technique for coordinated resource partitioning of the database server buffer pool, the storage server cache, and the storage server disk bandwidth. We focus on building a simple performance model in order to guide the search, by providing a good approximation of the overall solution. The performance model provides a resource-to-performance mapping for each application, in all possible resource quota configurations.
Our key ideas are to incorporate readily available information about the application and system into the performance model, and then refine the model through limited experimental sampling of actual behavior.

2.5 Validating System Performance

Modern systems are becoming increasingly large and complex consisting of several tiers, making it challenging for the administrator/user to understand the overall behavior of the system and diagnose performance problems. The ability to deploy automated tools to monitor large scale multi-tier systems has thus become a life-saving goal for the computer industry. Indeed, many commercial tools for coordinated monitoring and control of large scale systems exist; for example, HP’s Openview and IBM’s Tivoli products collect and aggregate information from a variety of sources and present this information graphically to operators. Nevertheless, the complexity of deployed systems exceeds the ability of humans to diagnose and respond to problems rapidly and correctly [53].

The traditional approach to automated problem detection is to develop analytical models of system structure and behavior, which may be represented quantitatively or as a set of event-condition-action rules [46]. These models may be costly to build automatically or may require extensive knowledge about the system. If specialized domain knowledge is used, then these models may either be incomplete, hence inaccurate, or may become obsolete as the system changes or encounters unprecedented situations. In contrast, recent research has investigated developing statistical models for automatic fault detection. These techniques derive probabilistic relationships, called functional invariants or correlations, between metrics captured at different points across the system [21, 38, 47]. The advantage of these approaches is that they are generic and need little or no domain knowledge. The system tracks these functional invariants and raises a user-level alarm whenever it detects a significant change in one or more of these functional invariants. These approaches can be applied to a wide variety of systems and can adapt rapidly to system changes. On the downside, they may trigger unacceptable levels of false alarms for benign changes, such as a workload mix change or an environmental setting change. Moreover, when a fault occurs, even if the fault is localized to one component, the metrics collected at many other components may show abnormal functional correlations. A ranking of the invalid invariants can provide some insight but the process of simply ranking and presenting correlations fails to provide a cohesive picture of the system to the user.
Rather than relying on the system to automatically find performance bottlenecks and diagnose anomalies, we argue that better insights can be gathered by initiating a *dialogue* between the administrator and the system. Our approach requires only high-level information about the system, which typically already exists and can be easily specified by the administrator. The ability to associate semantic meaning to system components, environment or load inputs, respectively, is crucial for an accurate and meaningful performance diagnosis. Expert knowledge about the *high-level* relationships between system components, called *prior beliefs*, is desirable for enhancing the ability to distinguish relevant events from noise and for filtering out false alarms.
Chapter 3

Optimizing Application Performance

This chapter describes the design, implementation, and evaluation of a context-aware prefetching algorithm called QuickMine.

3.1 Introduction

In many of today’s applications, such as, e-commerce, on-line stores, file utilities, photo galleries, etc., access to storage constitutes the major cost of processing a user request. Therefore, recent research has focused on techniques for alleviating the storage access latency through storage caching [30, 39, 60] and prefetching techniques [61, 62, 92, 93, 94].

Many traditional storage prefetching algorithms implement sequential prefetching, where the storage server prefetches a batch of sequential blocks upon detecting a sequential access pattern. Recent algorithms, like C-Miner* [61, 62], capture repeatable non-sequential access patterns as well. However, the storage system receives interleaved requests originating from many concurrent application streams. Thus, even if the logical I/O sequence of a particular application translates into physically sequential accesses, and/or the application pattern is highly repeatable, this pattern may be hard to recognize at the storage system. This is the case for concurrent execution of several applications sharing a network-attached storage, e.g., as shown in Figure 3.1, and also for a single application with multiple threads exhibiting different access patterns, e.g., a database application running multiple queries, as also shown in the figure.
We investigate prefetching in storage systems and present a novel caching and prefetching technique that exploits logical application contexts to improve prefetching effectiveness. Our technique employs a context tracking mechanism, as well as a lightweight frequent sequence mining technique \[3\]. The context tracking mechanism captures application contexts in an \textit{application independent manner}, with minimal instrumentation. These contexts are leveraged by the sequence mining technique for detecting block access patterns. We simply tag each application I/O block request with a context identifier corresponding to the higher level application context, e.g., a web interaction, database transaction, application thread, or database query, where the I/O request to the storage manager occurs. Such contexts are readily available in any application and can be easily captured. We then pass this context identifier along with each read block request, through the operating system, to the storage server. This allows the storage server to correlate the block accesses that it sees into frequent block sequences according to their higher level context. Based on the derived block correlations, the storage cache manager then issues block prefetches per context rather than globally.

At the storage server, correlating block accesses is performed by the frequent sequence mining component of our approach. In particular, we design and implement a lightweight and dynamic frequent sequence mining technique, called \textit{QuickMine}. Just like state-of-the-art prefetching algorithms \[61, 62\], \textit{QuickMine} detects sequential as well as non-sequential correlations using a history-based mechanism. \textit{QuickMine}'s key novelty lies in detecting and leveraging block correlations within logical application contexts. In addition, \textit{QuickMine} generates and adapts block correlations \textit{incrementally, on-the-fly}, through a lightweight mining algorithm. As we will show in our experimental evaluation, these novel features make \textit{QuickMine} uniquely suitable for on-line pattern mining and prefetching by (i) substantially reducing the footprint of the block correlations it generates, (ii) improving the likelihood that the block correlations maintained will lead to accurate prefetches and (iii) providing flexibility to dynamic changes in the application pattern, and the degree of concurrency.

We implement \textit{QuickMine} in our virtual storage prototype, \textit{Akash}. We also implement several alternative approaches for comparison with our scheme, including a baseline LRU cache replacement algorithm with no prefetching, and the following state-of-the-art context-oblivious prefetching schemes: two adaptive sequential prefetching schemes \[28, 44\] and the recently proposed \textit{C-Miner}∗ storage prefetching algorithm \[61, 62\].

In our experimental evaluation, we use three standard database applications: the TPC-W e-commerce benchmark, the RUBiS auctions benchmark and DBT-2 \[110\], a TPC-C-like
Figure 3.1: **Interleaved Accesses.** We show that the I/O accesses issued by the application are interleaved due to context-switching at the operating system and at the storage server. We show two databases connected to a shared storage server where each database has several threads handling transactions. Due to operating system context-switching, the I/Os from each thread are interleaved. In addition, the I/Os issued by one machine are interleaved with requests from another machine.

The applications have a wide range of access patterns. TPC-W and RUBiS are read-intensive workloads with highly repeatable access patterns; they contain 80% and 85% read-only transactions, respectively, in their workload mix. In contrast, DBT-2 is a write-intensive application with rapidly changing access patterns; it contains only 4% read-only transactions in its workload mix. We instrument the MySQL/InnoDB database engine to track the contexts of interest. We find that changing the DBMS to incorporate the context into an I/O request is trivial; the DBMS already tracks various contexts, such as database thread, transaction or query, and these contexts are easily obtained for each I/O operation. We perform experiments using our storage cache deployed within Akash in a networked storage environment.

Our experiments show that the context-aware QuickMine brings substantial latency reductions of up to factors of 2.0. The latency reductions correspond to reductions of miss-ratios in the storage cache of up to 60%. In contrast, the context oblivious schemes perform poorly for all benchmarks, with latencies comparable to, or worse than the baseline. This is due to either (i) inaccurate prefetches or (ii) non-repeatable (false) block correlations at
context boundaries, hence useless prefetch rules in the context-oblivious approaches. Our evaluation shows that QuickMine generates substantially more effective block correlation rules overall, in terms of both the number of prefetches triggered and the prefetching accuracy. We also show that QuickMine is capable of adjusting its correlation rules dynamically, without incurring undue overhead for rapidly changing patterns.

The rest of this chapter is organized as follows. Section 3.2 provides the necessary background and motivates our dynamic, context-aware algorithm. Section 3.3 introduces our QuickMine context-aware prefetching solution. We provide a detailed example in Section 3.4. Section 3.5 provides details of our implementation. Section 3.6 describes our experimental platform, methodology, other approaches in storage cache management that we evaluate in our experiments. Section 3.7 presents our experimental results and Section 3.8 discusses different factors affecting the performance of the prefetching algorithms. Section 3.9 concludes the chapter.

### 3.2 Motivation

We focus on improving the cache hit-ratio at the storage cache in a networked storage environment through prefetching. Our techniques are applicable to situations where the working set of storage clients, like a database system or file system, does not fit into the storage cache hierarchy i.e., into the combined caching capabilities of storage client and server. This situation is, and will remain common in the foreseeable future due to the following reasons.

First, while both client and storage server cache sizes are increasing, so are the memory demands of storage applications e.g., very large databases. Second, previous research has shown that access patterns at the storage server cache typically exhibit long reuse distances, hence poor cache utilization [22, 63]. Third, due to server consolidation trends towards reducing the costs of management in large data centers, several applications typically run on a cluster with consolidated storage, or even on the same physical server. This creates application interference, hence potential capacity misses, and reduces prefetching effectiveness in the shared storage-level cache [52]. In the following, we motivate our context-aware prefetching approach through a workload characterization for two e-commerce applications.

We analyze the access patterns of two popular e-commerce benchmarks: TPC-W and RUBiS. We conduct experiments using a network storage server, Akash, using the NBD (network block device) protocol built into Linux. The Akash server manages access to physical
storage and provides virtual block devices to applications. We experiment with each benchmark separately, varying the number of clients from 1 to 100. Each application runs on MySQL/InnoDB, which uses the NBD client kernel driver to mount the virtual block device. We provide more details on our experimental setup in Section 3.6.

We characterize the access patterns of the two applications using the following metrics. The average/maximum sequential run length [109] is the average/maximum length of physically sequential block sequences for the duration of the application’s execution. The average context access length is the average number of I/O requests issued by a logical unit of work in the application, i.e., by a transaction. Finally, the interference count [52] measures the interference in the storage cache, defined as the number of requests from other transactions that occur between consecutive requests from a given transaction stream.

Before going through our benchmark results, we will first present a simple example to further illustrate the metrics defined above. In this example, we compute the above metrics for the access stream for the access patterns at the storage shown in Figure 3.1. It shows a storage server hosting two virtual volumes mounted by two MySQL/InnoDB servers, each running two different threads, and each thread handling a separate client connection. Focusing on the first volume (i.e., the one to the left), we see that the interleaved access trace at the storage contains 5 sequential runs to that volume, where a sequential run is a sequence of I/Os issued to a contiguous set of data blocks. Specifically, five sequential runs exist for the first volume, i.e., \{(1, 2), (5), (7), (3), (9)\}, with average run length $\frac{2+1+1+1+1}{5} = 1.2$ and the maximum run length of 2. Indeed, the sequential runs for the de-tangled trace [109], i.e., separating the accesses by thread, as if each was running in isolation, would contain longer sequential runs, e.g., \{(1, 2, 3)\} as produced by accessing volume 1. This shows that interleaving reduces the average and maximum length of sequential runs. In the same example, in the overall access trace seen by the consolidated storage server, the accesses to volume 1 are interrupted 4 times by accesses to volume 2. The maximum interference i.e., gap between two subsequent accesses for volume 1 caused by interference from volume 2 is equal to 2 (corresponding to accesses \{V, X\}) and the average interference count is $\frac{1+1+1+2}{4} = 1.25$.

In our experiments, we first compute the sequential run lengths when each thread is run in isolation i.e., on the de-tangled trace for each of the two benchmarks. The lengths are: 1.05 (average) and 277 (maximum) for TPC-W and 1.14 (average) and 64 (maximum) for RUBiS. We then measure the sequential run lengths on the interleaved access traces, while progressively increasing the number of clients. We find that the sequential run length
Figure 3.2: **Interference count.** We show the average interference count for two e-commerce benchmarks with increasing number of clients. Due to the operating system context-switching, the I/Os issued by one thread is *interleaved* with I/Os of other threads running concurrently. The degree of interleaving (defined as *interference count*) increases with more clients accessing the benchmark.

decreases significantly as we increase the concurrency degree. For example, with 10 concurrently running clients, the sequential run length is already affected: 1.04 (average) and 65 (maximum) for TPC-W, and 1.05 (average) and 64 (maximum) for RUBiS. With the common e-commerce workload of 100 clients, the average sequential run length asymptotically approaches 1.0 for both benchmarks. To further understand the drop in sequential run length, we plot the interference count for each benchmark when increasing the number of clients in Figure 3.2. The figure shows that the interference count increases steadily with the number of concurrent clients, from 5.87 for TPC-W and 2.91 for RUBiS at 10 clients, to 82.22 for TPC-W and 15.95 for RUBiS with 100 concurrently running clients.

To study the lower interference count in RUBiS compared to TPC-W, we compute the average *context access length* per transaction, in the two benchmarks. We find that the average context access length for RUBiS is 71 blocks, compared to 1223 blocks for TPC-W, 87% of the RUBiS transactions are short, reading only 1 to 10 blocks of data, compared to 79% in TPC-W, and several RUBiS transactions access a single block. Hence in TPC-W, longer logical access sequences result in higher interference opportunities, and for both benchmarks only a few logical sequences translate into physically sequential accesses.

The preliminary results presented in this section show that: (i) opportunities for sequen-
tial prefetching in e-commerce workloads are low, and (ii) random (non-repeatable) access interleavings can occur for the high levels of application concurrency common in e-commerce workloads. Accordingly, these results motivate the need for a prefetching scheme that (i) exploits generic (non-sequential) access patterns in the application, and (ii) is aware of application concurrency and capable of detecting access patterns per application context. For this purpose, in the following, we introduce our algorithm that employs data mining principles to discover access correlations at runtime in addition to a context tracking framework that allows the I/O accesses from multiple high-level contexts.

### 3.3 Context-aware Mining and Prefetching

In this section, we describe our approach of context-aware prefetching at the storage server. We first present an overview of our approach, and introduce the terminology we use. We then describe in more detail our technique for tracking application contexts, the QuickMine algorithm for discovering block correlations and discuss how we leverage them in our prefetching algorithm.

#### 3.3.1 Overview

We use application-level contexts to guide I/O block prefetching at the storage server. An application-level context is a logical unit of work corresponding to a specific level in the application’s structural hierarchy e.g., a thread, a web interaction, a transaction, a query template, or a query instance. We tag each I/O access with a context identifier provided by the application and pass these identifiers through the operating system to the storage server. This allows the storage server to group block accesses per application-level context. In the example in Figure 3.1, assuming that the context identifier of an I/O access is the thread identifier, the storage server is able to differentiate that blocks \{1, 2, 3\} are accessed by Thread-1 and blocks \{5, 7, 9\} are accessed by Thread-2 from the interleaved access pattern.

Within each sequence of block accesses thus grouped by application context, the storage server applies our frequent sequence mining algorithm, called QuickMine. The QuickMine algorithm predicts a set of blocks to be accessed with high probability in the near future. The predictions are made based on mining past access patterns. Specifically, QuickMine derives per-context correlation rules for blocks that appear together frequently for a given context. Creation of new correlation rules and pruning useless old rules for an application and its various contexts occurs incrementally, on-the-fly, while the system performs its regu-
lar activities, including running other applications. The *QuickMine* algorithm is embedded in the storage server cache. The storage cache uses the derived rules to issue prefetches for blocks that are expected to occur within short order after a sequence of already seen blocks.

**Terminology** For the purposes of our data mining algorithm, a *sequence* is a list of I/O reads issued by an application context ordered by the time of their disk requests. A sequence database

$$\mathcal{D} = \{S_1, S_2, \ldots, S_n\} \tag{3.1}$$

is a set of sequences. The *support* of a sequence $R$ in the database $\mathcal{D}$ is the number of sequences for which $R$ is a *subsequence*. A subsequence is considered *frequent* if it occurs with a frequency higher than a predefined *min-support* threshold.

Blocks, i.e., the segments of disk data referred in I/O reads, in frequent subsequences are said to be *correlated*. Correlated blocks do not have to occur consecutively in a sequence but they should occur within a small distance, called a *gap* or *lookahead distance*, denoted $G$. The larger the lookahead distance, the more aggressive the algorithm is in determining correlations. For the purposes of our storage caching and prefetching algorithm, we classify cache accesses into one of three result categories. A *cache hit* is an application demand access to a block currently in the storage cache. A *prefetch hit* is a block demand access for which a prefetch has been previously issued; the respective block may or may not have arrived at the cache at the time of the demand access. All other accesses are cache *misses*.

### 3.3.2 Tracking High-Level Contexts

A *context* for an I/O request seen at the storage server is the application-level unit of work that issued the I/O request. We use our context tracking for database applications and track information about three types of contexts with different granularities: *application thread*, *database transaction* and *database query*. Contexts are delineated with begin and end delimiters and can be nested. For *application thread* contexts, we tag block accesses with the process/thread identifier of the database system thread running the interaction. Tracking accesses by application thread is coarse-grained as a thread may execute several transactions and queries. A finer-grained context is the *database transaction* context where we tag all block accesses between the *BEGIN* and *COMMIT/ABORT* with the transaction identifier. The finest-grained context is the *database query* context where we associate each block access
with the query or query template identifier. In addition, for database queries, we can track
the context of each query instance, or of each query template i.e., the same query with
different argument values.

While defining meaningful contexts is intuitive, defining the right context granularity for
optimizing the prefetching algorithm may be non-trivial. There is a trade-off between using
coarse-grained contexts and fine-grained contexts. Fine-grained contexts provide greater
prediction accuracy, but may limit prefetching aggressiveness because they contain fewer
accesses. Coarse-grained contexts, on the other hand, provide more prefetching opportuni-
ties, but lower accuracy due to more variability in access patterns, e.g., due to control flow
within a transaction or thread.

We study the feasibility of our tagging approach in three open-source database engines:
MySQL, PostgreSQL, and Apache Derby, and we find the necessary changes to be trivial
in all these existing code bases. In each database engine, we simply reuse pre-existing
begin and end markers, such as, connection establishment/connection tear-down, BEGIN
and COMMIT/ROLLBACK statements, and thread creation and destruction to identify the start
and end of a context. The implementation and results presented are based on minimal
instrumentation of the MySQL/InnoDB database server to track transaction and query
template contexts.

3.3.3 Determining Blocks to Prefetch

QuickMine derives block correlation rules for each application context as follows. Given a se-
quence of already accessed blocks \( \{a_1, a_2, \ldots, a_k\} \), and a lookahead parameter \( G \), QuickMine
derives block correlation rules of the form

\[
\{a_i \& a_j \rightarrow a_k\}
\]

for all \( i, j \) and \( k \), where \( \{a_i, a_j, a_k\} \) is a subsequence and \( i < j < k \), \( (j - i) < G \) and
\( (k - i) < G \).

For each rule of the form \( \{a \& b \rightarrow c\} \), \( \{a \& b\} \) is called a sequence prefix, and represents
two blocks already seen on a hot path through the data i.e., a path taken repeatedly during
the execution of the corresponding application context. For the same rule, \( \{c\} \) is one of
the block accesses about to follow the prefix with high probability and is called a sequence
suffix. Typically, the same prefix has several different suffixes depending on the lookahead
distance \( G \) and on the variability of the access patterns within the given context, e.g., due
to control flow. For each prefix, we maintain a list of possible suffixes, up to a cut-off max-suffix number. In addition, with each suffix, we maintain a frequency counter to track the number of times the suffix was accessed i.e., the support for that block. The list of suffixes is maintained in order of their respective frequencies to help predict the most probable block(s) to be accessed next. For example, assume that QuickMine has seen access patterns \{(a_1, a_2, a_3, a_4), (a_2, a_3, a_4), (a_2, a_3, a_5)\} in the past for a given context. QuickMine creates rules \{a_2&a_3 \rightarrow a_4\} and \{a_2&a_3 \rightarrow a_5\} for this context. Further assume that the current access sequence matches the rule prefix \{a_2&a_3\}. QuickMine predicts that the next block to be accessed will be suffix \{a_4\} or \{a_5\} in this order of probability because \{a_2, a_3, a_4\} occurred twice while \{a_2, a_3, a_5\} occurred only once.

We track the blocks accessed within each context and create/update block correlations for that context whenever a context ends. We maintain all block correlation rules in a rule cache, which allows the pruning of old rules through simple cache replacement policies. The cache replacement policies act in two dimensions: (i) for rule prefixes and (ii) within the suffixes of each prefix. We keep the most recent max-prefix prefixes in the cache. For each prefix, we keep max-suffix most probable suffixes. Hence, the cache replacement policies are LRU (Least-Recently-Used) for rule prefixes and LFU (Least-Frequently-Used) for suffixes. Intuitively, these policies match the goals of our mining algorithm well. Since access paths change over time as the underlying data changes, we need to remember recent hot paths and forget past paths. Furthermore, as mentioned before, we need to remember only the most probable suffixes for each prefix. To prevent quick evictions, newly added suffixes are given a grace period.

### 3.3.4 Issuing Prefetch Requests

The storage cache uses the block correlation rules to issue block prefetches for predicted future read accesses. Block prefetches are issued upon a read block miss. We use the last two accesses of the corresponding context to search the rule cache for the prefix just seen in the I/O block sequence. We determine the set of possible blocks to be accessed next as the set of suffixes stored for the corresponding prefix. We prefetch blocks that are not currently in the cache starting with the highest support block up to either the maximum suffixes stored or the maximum prefetching degree.

The number of block prefetches issued upon a block miss called prefetch aggressiveness, is a configurable parameter. We set the prefetching aggressiveness to the same value as max-suffix for all contexts. However, in heavy load situations, we limit the prefetching ag-
gressiveness to prevent saturating the storage bandwidth. Specifically, we leverage our context information to selectively throttle or disable prefetching for contexts where prefetching is not beneficial. Prefetching benefit is determined per context, as the ratio of prefetches issued versus prefetched blocks used by the application.

The prefetched blocks brought in by one application context may be, however, consumed by a different application context due to data affinity. We call such contexts **symbiotic contexts**. We determine **symbiotic contexts** sets by assigning context identifiers tracking the issuing and using contexts for each prefetched cache block. We determine **symbiotic contexts** as follows. We add two variables to each block in the cache, `prefetchedBy` and `usedBy` indicating the query template of the thread issuing the prefetch request and the query template of the thread using the prefetched block. We maintain a sparse correlation matrix $C(N, N)$ where $N$ is the number of query templates we have in the system and we capture the relationship between different query templates. Each entry $C(i, j)$ in the matrix records a number between 0.0 and 1.0 showing the fraction of requests issued by template $i$ and consumed by $j$. If $C(i, j) > 0.5$ then we classify these two templates as symbiotic pairs. We merge the counts of template $i$ and $j$ and replace with a new identifier $k$ representing the symbiotic pair of $i$ and $j$. We then monitor the prefetching benefit at the level of **symbiotic context sets** rather than per individual context. We disable prefetching for contexts (or symbiotic context sets) performing poorly.

### 3.4 Example of Context-Aware Prefetching

We show an example of context-aware prefetching in Figure 3.4 using a sample database table consisting of Turing award winners (shown in Figure 3.3). The example illustrates how incorrect block correlations can be generated by a context-oblivious mining algorithm due to the different interleavings of I/O requests issued by different contexts. We characterize these incorrect correlations as a (i) **false rule** – where the rule prefix is incorrect and (ii) **false prefetches** – where the rule suffix is incorrect. In some cases, the correlation may contain both false rules and false prefetches. Both false rules and false prefetches lead to low prefetching efficiency as false rules do not trigger prefetches and false prefetches trigger prefetches for blocks that may not be accessed in the future. For simplicity, let us assume that each row in the table uses a different block on disk and also assume that there is an index on the `Firstname` and `Lastname` columns. On this table, we can issue several queries as shown in Figure 3.3. Let us assume that queries $Q_1$ and $Q_3$ are issued concurrently and
<table>
<thead>
<tr>
<th>Row</th>
<th>Year</th>
<th>Firstname</th>
<th>Lastname</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1966</td>
<td>Alan</td>
<td>Perlis</td>
</tr>
<tr>
<td>2</td>
<td>1967</td>
<td>Maurice</td>
<td>Wilkes</td>
</tr>
<tr>
<td>3</td>
<td>1968</td>
<td>Richard</td>
<td>Hamming</td>
</tr>
<tr>
<td>4</td>
<td>1969</td>
<td>Marvin</td>
<td>Minsky</td>
</tr>
<tr>
<td>5</td>
<td>1970</td>
<td>James</td>
<td>Wilkinson</td>
</tr>
<tr>
<td>6</td>
<td>1971</td>
<td>John</td>
<td>McCarthy</td>
</tr>
<tr>
<td>7</td>
<td>1972</td>
<td>Edsger</td>
<td>Dijkstra</td>
</tr>
<tr>
<td>8</td>
<td>1973</td>
<td>Charles</td>
<td>Bachman</td>
</tr>
<tr>
<td>9</td>
<td>1974</td>
<td>Donald</td>
<td>Knuth</td>
</tr>
</tbody>
</table>

Figure 3.3: Turing Award Winners. To compare context-aware prefetching and context-unaware prefetching algorithms, we use the following table consisting of Turing award winners and show the rows that are accessed for three sample queries.
Figure 3.4: **Walkthrough.** We compare the *QuickMine* algorithm with a context-oblivious prefetching algorithm. It shows the *false* correlations that are generated for an interleaved traces of I/O accesses from two concurrently running transactions with a context-oblivious algorithm.
the resulting I/Os are interleaved as shown in Figure 3.4 and we use a mining algorithm with a \textit{lookahead gap} of 5.

Let us consider a context-oblivious mining algorithm first. The initial set of correlations for the access prefix \{1&3\} is highlighted in Figure 3.4. With context-oblivious mining, the mined patterns are generated based on the \textit{specific} interleaving of accesses. In this case, this leads the algorithm to discover correlations \{1&3\} → \{2,7,8,9\}, \{1&2\} → \{7,8,9\} and \{1&7\} → \{8,9\} within the highlighted area. This shows the incorrect correlations generated by context-oblivious mining algorithms where the first rule contains a \textit{false rule} and a \textit{false prefetch} where both the rule prefix (\{1&3\}) and the rule suffix (\{2,7,8,9\}) contain blocks from two different queries. In the second rule, the rule prefix is correct as blocks 1 and 2 are accessed by query \(Q_1\) however the the rule suffix contains blocks of \(Q_3\). Both false rules and false prefetches reduce the effectiveness of prefetching as false rules do not trigger prefetches and false prefetches trigger prefetches for incorrect blocks.

On the other hand, with context-aware prefetching, the algorithm is not affected by I/O interleavings of different threads thus discovers correct correlations – that is, it does not contain \textit{false rules} or \textit{false prefetches}. As shown in Figure 3.4, the correlations discovered are \{1&2\} → \{3,4,5\}, \{1&3\} → \{4,5\}, and \{1&4\} → \{5\} where each correlation contains only accesses from query \(Q_1\).

### 3.5 Prototype Implementation

Our infrastructure consists of a virtual storage system prototype, \textit{Akash}, designed to run on commodity hardware. It supports data accesses to multiple virtual volumes for any storage client, such as, database servers and file systems. It uses the Network Block Device (NBD) driver packaged with Linux to read and write logical blocks from the virtual storage system, as shown in Figure 3.5. NBD is a standard storage access protocol similar to iSCSI, supported by Linux. It provides a method to communicate with a storage server over the network. The client machine (shown in left) mounts the virtual volume as a NBD device (e.g., \texttt{/dev/nbd1}) which is used by MySQL as a raw disk partition, (e.g., \texttt{/dev/raw/raw1}). We modify existing \textit{client} and \textit{server} NBD protocol processing modules for the storage client and server, respectively, in order to interpose our storage cache and disk controller modules on the I/O communication path, as shown in the figure. To evaluate context-aware prefetching, we modify the MySQL database and the Linux operating system to support context-awareness. In the following, we first describe the interfaces and communication...
Figure 3.5: **Storage Architecture.** Our storage infrastructure consists of a server node and a client node. The server node runs Akash and exports multiple virtual storage volumes. The client node (running MySQL) connects to a virtual storage volume using the NBD driver in Linux.

between the core modules, then describe the role of each module in more detail.

### 3.5.1 Interfaces and Communication

Storage clients, such as MySQL, use NBD for reading and writing logical blocks. For example, as shown in Figure 3.5, MySQL mounts the NBD device (/dev/nbd1) on /dev/raw/raw1. The Linux virtual disk driver uses the NBD protocol to communicate with the storage server. In NBD, an I/O request from the client takes the form `<type,offset,length>` where `type` is a **read** or **write**. The I/O request is passed by the OS to the NBD kernel driver on the client, which transfers the request over the network to the NBD protocol module running on the storage server.

### 3.5.2 Modules of the Akash Storage Server

Each module of the Akash storage server consists of several threads processing requests. The modules are interconnected through in-memory bounded buffers. The modular design allows us to build many storage configurations by simply connecting different modules together.

> **Disk Module:** The disk module sits at the lowest level of the module hierarchy. It provides the interface with the underlying physical disk by translating application I/O requests to the virtual disk into `pread()`/ `pwrite()` system calls, reading/writing the
<table>
<thead>
<tr>
<th>Akash Storage System</th>
<th>9431</th>
</tr>
</thead>
<tbody>
<tr>
<td>Headers</td>
<td>1493</td>
</tr>
<tr>
<td>Core</td>
<td>572</td>
</tr>
<tr>
<td>Modules</td>
<td>2499</td>
</tr>
<tr>
<td>Buffer Pool</td>
<td>868</td>
</tr>
<tr>
<td>Other Data Structures</td>
<td>2450</td>
</tr>
<tr>
<td>Prefetching Algorithms</td>
<td>533</td>
</tr>
<tr>
<td>Monitoring</td>
<td>1016</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Resource Controller</th>
<th>3852</th>
</tr>
</thead>
<tbody>
<tr>
<td>Framework</td>
<td>3035</td>
</tr>
<tr>
<td>Algorithms</td>
<td>817</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Changes to Existing Components (Number of Files)</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>3</td>
</tr>
<tr>
<td>MySQL</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Miscellaneous Scripts and Test Programs</th>
<th>2561</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPC-C Benchmark</td>
<td>1014</td>
</tr>
<tr>
<td>IO Bench</td>
<td>1547</td>
</tr>
</tbody>
</table>

Table 3.1: Programming Effort. We show the programming effort needed to implement the different components that compose our multi-tier storage infrastructure. For evaluating changes made to existing components, we list the number of files modified rather than the number of lines.
underlying physical data. We disable the operating system buffer cache by using direct
I/O i.e., the I/O \texttt{O\_DIRECT} flag in Linux.

\begin{itemize}
\item \textbf{Quanta Module:} We use a quanta-based scheduling algorithm to proportionally allocate
the disk bandwidth to different workloads in our virtual storage system. Each workload
is given a quantum of time during which it uses the disk exclusively. This offers a strong
isolation between workloads as there are no \textit{disk seeks} between I/Os of two workloads. It
works as follows. When a workload is given a quantum, we first determine the number
of requests we can issue to disk such that they complete within the workload’s quantum.
To compute this value, we maintain an exponentially weighted average of the disk service
time and the application’s concurrency level. Using these two values, we compute the
number of requests that can be issued per workload such that all requests finish within
the quantum. First, we issue requests that were enqueued while waiting for the quantum
to begin. Then, we issue requests that arrive during the scheduling quantum. We stop
issuing requests if we determine that by issuing a request, we will exceed the workload’s
quantum. In this case, new requests will be enqueued until the next quantum for this
workload. This module is used to implement dynamic partitioning of the disk bandwidth.
It is presented in more detail in Chapter 4.

\item \textbf{Cache Module:} The cache module supports context-aware caching and prefetching. We
developed a portable caching library providing a simple hashtable-like interface modelled
after \textit{Berkeley DB}. If the requested block is found in the cache, the access is a \textit{cache hit}
and we return the data. Otherwise, the access is a \textit{cache miss}, we fetch the block from
the next level in the storage hierarchy, store it in the cache, then return the data.

When prefetching is enabled, the cache is partitioned into two areas: a main cache (MC)
and a prefetch cache (PFC). The PFC contains blocks that were fetched from disk by
the prefetching algorithm. The MC contains blocks that were requested by application
threads. If an application thread requests a block for which a prefetch has been issued,
we classify the access as a \textit{prefetch hit}. A block \textit{prefetch hit} may imply either waiting
for the block to arrive from disk, or simply moving the block from PFC to MC if the
block is already in the cache. We use \textit{Berkeley DB} to implement the rule cache and
store the mined correlations. The caching keys are rule \emph{prefixes} and the cached data are
the rule \emph{suffixes}. The suffixes are maintained using the LFU replacement algorithm and
the prefixes are maintained using LRU. The LRU policy is implemented using \textit{timeouts},
where we periodically purge old entries. We configure \textit{Berkeley DB}’s environment to use
\end{itemize}
int ctx_pread(int fd, off_t offset, size_t len, ctx_t *ctx);
int ctx_pwrite(int fd, off_t offset, size_t len, ctx_t *ctx);
int ctx_action(int fd, int action, ctx_t *ctx);

Listing 3.1: New Linux System Calls. We create three new system calls to inform the storage server of the begin/end of contexts and to provide context-aware versions of the read/write system calls.

We create three new system calls to inform the storage server of the begin/end of contexts and to provide context-aware versions of the read/write system calls.

a memory pool of 4MB.

▷ NBD Protocol Module: We modify the original NBD protocol module on the server side, used in Linux for virtual disk access, to convert the NBD packets into our own internal protocol packets, i.e., into calls to our server cache module.

3.5.3 Changes made to Existing Code

To allow for context awareness, we make minor changes to MySQL, the Linux kernel, and the NBD protocol to piggyback context information throughout the I/O stack.

▷ Linux: The changes required in the kernel are straightforward and minimal. In the simplest case, we need to pass a context identifier on I/O calls as a separate argument into the kernel. In order to allow more flexibility in our implementation, and enhancements such as per-context tracking of prefetch effectiveness, we pass a handle to a context structure, which contains the transaction identifier, and query template identifier. We add three new system calls, ctx_pread(), ctx_pwrite() and ctx_action() that allow an additional context parameter to be passed to the storage server. ctx_action() allows us to inform the storage server of context begin/end delimiters. Each system call takes a ctx_t * as a parameter representing the context of the I/O call, as shown in Listing 3.1. This context handle is passed along the kernel until it reaches the lowest level where the kernel contacts the block storage device. Specifically, we add a field ctx to struct request, which allows us to pass the context information through the I/O subsystem with no additional code changes. Once the I/O request reaches the NBD driver code, we copy the context information into the NBD request packet and pass the information to the storage server.

▷ NBD Protocol: We simply piggyback the context information on the NBD packet. In addition, we add two new messages to the NBD protocol, for the corresponding system call
ctx_action(), to signify the beginning of a context (CTX_BEG) and the end of a context (CTX_END).

**MySQL:** MySQL uses a multi-threaded architecture to manage client connections. The execution context of each client connection is encapsulated in a THD object. For example, THD.query contains the query currently being executed by the thread. We generate the query template identifier using the query string. In addition, we call our ctx_action() as appropriate, e.g., at transaction begin/end and at connection setup/tear-down to inform the storage server of the start/end of a context.

### 3.6 Evaluation

In this section, we describe several prefetching algorithms we use for comparison with QuickMine and evaluate the performance using three industry-standard benchmarks: TPC-W, RU-BiS, and DBT-2.

#### 3.6.1 Prefetching Algorithms used for Comparison

We use several prefetching algorithms for comparison with QuickMine. These algorithms are categorized into sequential prefetching schemes (RUN and SEQ) and history based prefetching schemes (C-Miner*). The RUN and C-Miner* algorithms share some of the features of QuickMine, specifically, some form of concurrency awareness (RUN) and history-based access pattern detection and prefetching (C-Miner*).

**Adaptive Sequential Prefetching**

We implement an adaptive sequential prefetching scheme that is similar to read-ahead algorithm implemented in Linux [28]. In this algorithm prefetching is activated when the algorithm detects a sequence (S) of accesses to K contiguous blocks. When the sequence is detected, the algorithm creates two windows: a current window and a read-ahead window each 32KB in size. Prefetches are issued for blocks contained in both windows. In addition, the number of prefetched blocks used in the current window (denoted as f) is monitored.

When a block is accessed in the readahead window, the current window is set to the readahead window and a new read-ahead window of size is 2f is created. To limit the prefetching aggressiveness, the size of the readahead window is limited through two parameters: min and max. Specifically, if 2f < min, prefetching is stopped for sequence S. If 2f > max,
window size is limited to $max$. We set $K$ to 32KB, $min$ to 32KB, and $max$ to 128KB as suggested by the authors [28].

**Run-Based Prefetching**

Hsu et al. [44] show that many workloads, particularly database workloads, do not exhibit strict sequential behavior, mainly due to high application concurrency. To capture sequentiality in a multi-threaded environment, Hsu et al. introduce *run-based prefetching* (RUN) [45]. To capture sequential accesses by concurrent threads, the *run-based prefetching* maintains a set of possible sequences (*runs*) in a stack $L$. A reference $r$ to block $b$ is considered to be part of a sequential run $R$ if $b$ lies within $-extent_{backward}$ and $+extent_{forward}$ of the largest block accessed in $R$, denoted by $R_{maxBlock}$. This modified definition of sequentiality thus accommodates small jumps and small reverses within an access trace. Once the size of the run $R$ exceeds a sequentiality threshold (32KB), prefetching is initiated for 16 blocks from $R_{maxBlock} + 1$ to $R_{maxBlock} + 16$.

In addition, each run $R$ maintains a *run count* ($R.runCount$) to keep track of the number of accesses to unique blocks that fall within the sequential run. The *run* stack is maintained as follows. On a block access, the stack is searched to find a matching run $R$. A run $R$ is a matching run if $b > (R_{maxBlock} - extent_{backward})$ and $b < (R_{maxBlock} + extent_{forward})$. After the match, the *run count* ($R.runCount + +$) is incremented and the run $R$ is moved to the top of the stack $S$. If the *run count* exceeds a sequentiality threshold, *seqThreshold* (32KB), prefetching is limited to 16 blocks, from $R_{maxBlock} + 1$ to $R_{maxBlock} + 16$. If no matching run is found, the run from the bottom of the stack is evicted and we add a new run $R_{new}$, with parameters initialized to $R_{new}.maxBlock = b$ and $R_{new}.runCount = 1$. We set *seqThreshold* = 32KB, $extent_{forward} = 16$, $extent_{backward} = 8$ as suggested by the authors. In addition, we set the run stack size, $|L| = 1024$, to accommodate sequential accesses by all storage threads.

**History-based Prefetching**

Several history based prefetching algorithms have been proposed in recent work [36, 45, 56, 63]. *C-Miner* is a static mining algorithm that extracts frequent block subsequences by mining the entire sequence database [62].

*C-Miner* builds on the subsequence mining algorithm *CloSpan* [113]. It differs from *QuickMine* by mining block correlations off-line, on a long sequence of I/O block accesses.
First, \textit{C-Miner}\textsuperscript{*} breaks the long sequence trace into smaller sequences and creates a sequence database. From these sequences, as in \textit{QuickMine}, the algorithm considers frequent sequences of blocks that occur within a \textit{gap} window. Given the sequence databases and using the \textit{gap} and \textit{min\_support} parameters, the algorithm extracts frequent \textit{closed} sequences, i.e., subsequences whose support value is different from that of its super-sequences. For example, if \{\textit{a}1,\textit{a}2,\textit{a}3,\textit{a}4\} is a frequent subsequence with support value of 5 and \{\textit{a}1,\textit{a}2,\textit{a}3\} is a subsequence with support value of 5 then, only \{\textit{a}1,\textit{a}2,\textit{a}3,\textit{a}4\} will be used in the final result.

On the other hand, if \{\textit{a}1,\textit{a}2,\textit{a}3\} has a support of 6 then, both sequences are recorded. For each \textit{closed} frequent sequence e.g., \{\textit{a}1,\textit{a}2,\textit{a}3,\textit{a}4\}, \textit{C-Miner}\textsuperscript{*} generates association rules of the form \{(\textit{a}1 \rightarrow \textit{a}2), (\textit{a}1 \& \textit{a}2 \rightarrow \textit{a}3), \ldots, (\textit{a}3 \rightarrow \textit{a}4)\}.

As an optimization, \textit{C-Miner}\textsuperscript{*} uses the frequency of a rule suffix in the rule set to prune predictions of low probability through a parameter called \textit{min\_confidence}. For example, if the mined trace contains 80 sequences with \{\textit{a}2\&\textit{a}3 \rightarrow \textit{a}4\} and 20 sequences with \{\textit{a}2\&\textit{a}3 \rightarrow \textit{a}5\}, then \{\textit{a}2\&\textit{a}3 \rightarrow \textit{a}5\} has a (relatively low) confidence of 20\% and might be pruned depending on the \textit{min\_confidence} threshold. In our experiments, we use \textit{max\_gap} = 10, \textit{min\_support} = 1, and \textit{min\_confidence} = 10\% for \textit{C-Miner}\textsuperscript{*}.

### 3.6.2 Benchmarks

We evaluate our techniques using industry-standard benchmarks: TPC-W, RUBiS, and DBT-2.

- **TPC-W**: The TPC-W benchmark from the Transaction Processing Council is a transactional web benchmark designed for evaluating e-commerce systems. Several web interactions are used to simulate the activity of a retail store. The database size is determined by the number of items in the inventory and the size of the customer population. We use 100K items and 2.8 million customers which results in a database of about 4 GB. We use the \textit{shopping} workload that consists of 20\% writes. To fully stress our architecture, we create TPC-W\textsuperscript{10} by running 10 TPC-W instances in parallel creating a database of 40 GB.

- **RUBiS**: We use the RUBiS Auction Benchmark to simulate a bidding workload similar to e-Bay. The benchmark implements the core functionality of an auction site: selling, browsing, and bidding. We distinguish between three kinds of user sessions: visitor, buyer, and seller. For a visitor session, users need not register but are only allowed to browse. Buyer and seller sessions require registration. In addition to the functionality provided during the visitor sessions, during a buyer session, users can bid on items and
consult a summary of their current bid, rating, and comments left by other users. We are using the default RUBiS bidding workload containing 15% writes, considered the most representative of an auction site workload according to an earlier study of e-Bay workloads [88]. We create a scaled workload, RUBiS$^{10}$ by running 10 RUBiS instances in parallel, creating a database of 14GB.

TPC-C/DBT-2: The TPC-C benchmark [80] simulates a wholesale parts supplier that operates using a number of warehouse and sales districts. It simulates a wholesale parts supplier that operates using a number of warehouse and sales districts. Each warehouse has 10 sales districts and each district serves 3000 customers. The workload involves transactions from a number of terminal operators centered around an order entry environment. We scale TPC-C by using 128 warehouses, which gives a database footprint of 32GB. For the evaluation of context-aware prefetching, we use the DBT-2 benchmark, an OLTP workload derived from the TPC-C benchmark [80, 110]. We scale DBT-2 by using 256 warehouses, which gives a database footprint of 60GB.

3.6.3 Methodology

We run our Web based applications on a dynamic content infrastructure consisting of the Apache web server, the PHP application server and the MySQL/InnoDB (version 5.0.24) database storage engine. For the database applications, we use the test harness provided by each benchmark while hosting the database on MySQL. We run the Apache Web server and MySQL on Dell PowerEdge SC1450 with dual Intel Xeon processors running at 3.0 Ghz with 2GB of memory. MySQL connects to the raw device hosted by the NBD server. We run the NBD server on a Dell PowerEdge PE1950 with 8 Intel Xeon processors running at 2.8 Ghz with 3GB of memory. To maximize IO bandwidth, we use RAID 0 on 15 10K RPM 250GB hard disks. We install Ubuntu 6.06 on both the client and server machines with Linux kernel version 2.6.18-smp.

We configure our caching library to use 16KB block size to match the MySQL/InnoDB block size. We use 100 clients for TPC-W and RUBiS. For DBT-2, we use 256 warehouses. We run each experiment for two hours. We train $C$-Miner$^*$ on a trace collected from the first hour of the experiment. We measure statistics for both $C$-Miner$^*$ and QuickMine during the second hour of the experiment. We use a lookahead value of 10 for $C$-Miner$^*$, which is the best value found experimentally for the given applications and number of clients used in the experiments. QuickMine is less sensitive to the lookahead value, and any lookahead
value between 5 and 10 gives similar results. We use a lookahead value of 5 for QuickMine.

### 3.7 Results

We evaluate the performance of the following schemes: a baseline caching scheme with no prefetching (denoted as LRU), adaptive sequential prefetching (SEQ), run-based prefetching (RUN), C-Miner*, and QuickMine. In this section, we present the overall performance of those schemes, whereas in Section 3.8, we provide detailed analysis to further understand the achieved overall performance of each scheme.

In this section, we measure the storage cache hit-ratios, miss-ratios, prefetch hit-ratios and the average read latency for each of the prefetching schemes by running our three benchmarks in several cache configurations. For all experiments, the MySQL/InnoDB buffer pool is set to 512MB and we partition the storage cache such that the prefetching area is fixed at 4% of the total storage cache size. For TPC-W and RUBiS, we use 100 clients and we vary the storage cache size, showing results for 512MB, 1024MB, and 2048MB storage cache for each benchmark. For DBT-2, we show results only for the 1024MB storage cache, since results for other cache sizes are similar. In QuickMine, we use query-template contexts for TPC-W and RUBiS and transaction contexts for DBT-2. However, the results vary only slightly with the context granularity for our three benchmarks.

#### 3.7.1 TPC-W E-Commerce Workload

Figures 3.6a-3.6c show the hit-ratios, miss-ratios, and prefetch hit-ratios for all prefetching schemes with TPC-W. For a 512MB storage cache, as shown in Fig. 3.6a, the baseline (LRU) miss-ratio is 89%. The sequential prefetching schemes reduce the miss-ratio by 5% on average. C-Miner* reduces the miss-ratio by 15%, while QuickMine reduces the miss-ratio by 60%. The benefit of sequential prefetching schemes is low due to the lack of sequentiality in the workload. With larger cache sizes, the baseline miss-ratios are reduced to 45% for the 1024MB cache (Fig. 3.6b) and to 20% for the 2048MB cache (Fig. 3.6c). With lower miss-ratios, there are lower opportunities for prefetching. In spite of this, QuickMine still provides a 30% and 17% reduction in miss-ratios for the 1024MB and 2048MB cache sizes, respectively.

For QuickMine, the cache miss reductions translate into substantial read latency reductions as well, as shown in Figure 3.6d. The reductions in the cache misses translate into
Figure 3.6: **TPC-W E-Commerce Workload.** We show the benefit of prefetching data for the TPC-W benchmark for three different storage cache sizes: 512MB, 1024MB, and 2048MB. Prefetching data reduces the miss-ratio at the storage server resulting in lower read latencies. With smaller caches the base (with no-prefetching) miss-ratio is high resulting in large improvements with prefetching.
decreases in overall storage access latency by factors ranging from 2.0 for the 512MB to 1.22 for the 2048MB caches, respectively. This is in contrast to the other prefetching algorithms, where the average read latency is comparable to the baseline average latency for all storage cache sizes.

### 3.7.2 RUBiS Auctions Workload

Context-aware prefetching benefits RUBiS as well, as shown in Figure 3.7. The baseline (LRU) cache miss ratios are 85%, 36%, and 2% for the 512MB, 1024MB, and 2048MB cache sizes, respectively. For a 512MB cache, as shown in Figure 3.7a, the SEQ and RUN schemes reduce the miss-ratio by 6% and 9%, respectively. C-Miner* reduces the miss-ratio by only 3% while QuickMine reduces the miss-ratio by 48%. In RUBiS, the queries access a spatially-local cluster of blocks. Thus, triggering prefetching for a weakly sequential access pattern, as in RUN, results in more prefetch hits than for SEQ. C-Miner* performs poorly for RUBiS because many RUBiS contexts are short. This results in many false correlations across context boundaries in C-Miner*. Hence, only a few prefetch rule prefixes derived during training can be later matched on-line, while many rule suffixes are pruned due to low confidence. QuickMine overcomes the limitations of C-Miner* by tracking correlations per context.

The performance of the prefetching algorithms is reflected in the average read latency as well. As shown in Figure 3.7d, the sequential prefetching schemes (SEQ and RUN) reduce the average read latency by up to 10% compared to LRU. The reductions in miss-ratio using QuickMine translate to reductions in read latencies of 45% (512MB) and 22% (1024MB) compared to LRU, corresponding to an overall storage access latency reduction by a factor of 1.63 for the 512MB cache and 1.3 for the 1024MB cache.

### 3.7.3 DBT-2 Transaction Processing Workload

Prefetching is difficult for DBT-2, since the workload mix for this benchmark contains a high fraction of writes; furthermore, some of its transactions issue very few I/Os. The I/O accesses are not sequential. As Figure 3.8 shows, the lack of sequentiality causes the sequential prefetching algorithms to perform poorly. Sequential prefetching schemes decrease the miss-ratio by less than 1%. C-Miner* and QuickMine perform slightly better. C-Miner* lowers the miss-ratio by 2% and QuickMine reduces the miss-ratio by 6%. However, the high I/O footprint of this benchmark causes disk congestion, hence increases the prefetch hit latency. Overall, the average read latency increases by 2% for the sequential prefetching
Figure 3.7: **RUBiS Auctions Workload.** We show the benefit of prefetching data for the RUBiS benchmark for three different storage cache sizes: 512MB, 1024MB, and 2048MB. Prefetching data reduces the miss-ratio at the storage server resulting in lower read latencies. With smaller caches the base (with no-prefetching) miss-ratio is high resulting in large improvements with prefetching.
Figure 3.8: DBT-2 Transaction Processing Workload. We show the benefit of prefetching for DBT-2 using a 1024MB storage cache. Prefetching is difficult since DBT-2 has many writes leading to changing access patterns. The average read latency increases by 2% for sequential prefetching schemes and C-Miner* while QuickMine reduces the latency by 3%.

schemes and C-Miner*, while the read latency is reduced by 3% for QuickMine.

3.8 Detailed Analysis

In this section, we evaluate the prefetching effectiveness of the different schemes by measuring the number of prefetches issued and their accuracy, i.e., the percentage of prefetched blocks consumed by the application. Both metrics are important since if only a few prefetches are issued, their overall impact is low, even at high accuracy for these prefetches. We also compare the two history-based schemes, QuickMine and C-Miner*, in more detail. Specifically, we show the benefits of context awareness and the benefit of incremental mining, versus static mining.

3.8.1 Detailed Comparison of Prefetching Effectiveness

In Figure 3.9, we show the number of prefetches issued, and their corresponding accuracy for all prefetching algorithms. For TPC-W with a 512MB cache, shown in Figure 3.9a, QuickMine issues 2M prefetches, while C-Miner*, SEQ, and RUN issue less than 500K prefetches. The RUN scheme is the least accurate (< 50%) since many prefetches are spuriously triggered. The SEQ scheme exhibits a better prefetch accuracy of between 50% and 75% for the three cache sizes. Both QuickMine and C-Miner* achieve greater than 75% accuracy. While C-Miner* has slightly higher accuracy than QuickMine for the rules issued, this accuracy
corresponds to substantially fewer rules than QuickMine. This is because many of the C-Miner* correlation rules correspond to false correlations at the context switch boundary, that are not triggered at runtime. As a positive side-effect, the higher number of issued prefetches in QuickMine allows the disk scheduler to re-order the requests for optimizing seeks, thus reducing the average prefetch latency. As a result, average prefetch hit latencies are significantly lower in QuickMine compared to C-Miner*, specifically, 600µs versus 2400µs for the 512MB cache. For comparison, a cache hit takes 7µs on average and a cache miss takes 3200µs on average, for all algorithms.

For RUBiS with a 512MB cache, shown in Figure 3.9c, QuickMine issues 1.5M prefetches, which is ten times more than C-Miner* and SEQ. In the RUN scheme, the spatial locality of RUBiS causes more prefetches (250K) to be issued compared to SEQ, but only 38% of these are accurate, as shown in Figure 3.9d. As before, while C-Miner* is highly accurate (92%) for the prefetches it issues, substantially fewer correlation rules are matched at runtime compared to QuickMine, due to false correlations. With larger cache sizes, there is less opportunity for prefetching, because there are fewer storage cache misses, but at all cache sizes QuickMine issues more prefetch requests than other prefetching schemes. Similar as for TPC-W, the higher number of prefetches results in a lower prefetch hit latency for QuickMine compared to C-Miner* i.e., 150µs versus 650µs for RUBiS in the 512MB cache configuration.

3.8.2 Benefit of Context Awareness

We compare the total number of correlation rules generated by frequent sequence mining, with and without context awareness. In our evaluation, we isolate the impact of context awareness from other algorithm artifacts, by running C-Miner* without rule pruning, on its original access traces of RUBiS and DBT-2, and the de-tangled access traces of the same. In the de-tangled trace, the accesses are separated by thread identifier, then concatenated. We notice an order of magnitude reduction in the number of rules generated by C-Miner*. Specifically, on the original traces, C-Miner* without rule pruning generates 8M rules and 36M rules for RUBiS and DBT-2, respectively. Using the de-tangled trace, C-Miner* without rule pruning generates 800K rules for RUBiS and 2.8M rules for DBT-2. These experiments show that context awareness reduces the number of rules generated, because it avoids generating false block correlation rules for the blocks at the context switch boundaries.

Another benefit of context-awareness is that it makes parameter settings in QuickMine insensitive to the concurrency degree. For example, the value of the lookahead/gap parameter correlates with the concurrency degree in context oblivious approaches, such as
Figure 3.9: **Prefetching Effectiveness.** We show the number of prefetches issued and the accuracy of the prefetches for all prefetching algorithms. With context-awareness, *Quick-Mine* is able to discover more correlations (i.e., more prefetches issued) and higher accuracy.
C-Miner*, i.e., the lookahead needs to be higher to capture correlations for a higher concurrency degree. In contrast, QuickMine’s parameters, including the value of the lookahead parameter, are independent of the concurrency degree; they mainly control the prefetch aggressiveness. While the ideal prefetch aggressiveness does depend on the application and environment, QuickMine has built-in dynamic tuning mechanisms that make it robust to overly aggressive parameter settings.

The ability to dynamically tune the prefetching decisions at run-time is yet another benefit of context-awareness. For example, in TPC-W, QuickMine automatically detects that the BestSeller and the symbiotic pair of Search and NewProducts benefit the most from prefetching, while other queries in TPC-W do not. Similarly, it detects that only the StockLevel transaction in DBT-2 benefits from prefetching. Tracking the prefetching benefit per context allows QuickMine to selectively disable or throttle prefetching for low performing query templates thus avoiding unnecessary disk congestion caused by useless prefetches. In particular, this feature allows QuickMine to provide a small benefit for DBT-2, while C-Miner* degrades the application performance.

3.8.3 Tradeoffs of different Context Granularities

While defining meaningful contexts is intuitive, defining the right context granularity for optimizing the prefetching algorithm may be non-trivial. There is a tradeoff between using coarse-grained contexts and fine-grained contexts. Fine-grained contexts provide greater prediction accuracy while coarse-grained contexts provide more prefetching opportunities.

Let’s compare using a fine-grained context such as a query versus a coarser grained context such as a transaction or Web interaction in a dynamic content application accessing a database back-end. For small queries, such as when the primary key is specified, the access pattern is to simply traverse down the B-Tree to reach the data page. In many cases, the index pages are cached in the buffer pool and the only miss seen at the storage server is a request to read the data block. If there is only one miss then, no prefetch rule will ever be generated or triggered for that context. This limits our prefetching aggressiveness because we may not be able to fetch several blocks at the same time. In contrast, coarse-grained contexts such as a database transaction or an application thread are well suited for aggressive prefetching. Since application threads for an application include more than one transaction, and each database transaction includes several read or write queries, we can derive correlations across several queries or several transactions. On the down side, having a coarser-grained context does not necessarily translate into higher prefetching accuracy if the
access pattern within this context shows a lot of variability. Variability may occur due to control flow in the application code of complex database transactions or application threads. Since a fine grain context typically has less variability, the accuracy of block correlations is correspondingly higher for fine grained versus coarse grained contexts.

We examine the effect of context granularity on prefetching performance by looking at the number of rules generated for each granularity and the prefetching accuracy. We notice a 15% increase in accuracy for DBT-2 when using query versus transaction contexts. However, the number of rules issued is higher for the transaction context, due to higher prefetch opportunities, leading to slightly higher prefetch effectiveness with coarser grained contexts. There is little change in prefetching effectiveness for RUBiS and TPC-W since most of their Web interactions contain only 1 query.

3.8.4 Benefit of Incremental Mining

We show the benefit of dynamically and incrementally updating correlation rules through the use of the LRU based rule cache as in QuickMine versus statically mining the entire sequence database to generate association rules as in C-Miner* [61, 62]. Figure 3.10 shows the number of prefetch hits, cumulatively, over the duration of the experiment for C-Miner*, C-Miner* with periodic retraining (denoted as C-Miner+) and QuickMine. For these experiments, we train C-Miner* on the de-tangled trace to eliminate the effects of interleaved I/O, hence
make it comparable with *QuickMine*. As Figure 3.10 shows, the change in the access patterns limits the prefetching effectiveness of *C-Miner*\(^*\), since many of its mined correlations become obsolete quickly. Thus, no new prefetch hits are issued after the first 10 minutes of the experiment. In *C-Miner*\(^+\), where we retrain *C-Miner*\(^*\) at the 10 minute mark, and at the 20 minute mark of the experiment, the effectiveness of prefetching improves. However, *C-Miner*\(^+\) still lags behind *QuickMine*, which adjusts its rules continuously, on-the-fly. By dynamically aging old correlation rules and incrementally learning new correlations, *QuickMine* maintains a steady performance throughout the experiment. The dynamic nature of *QuickMine* allows it to automatically and gracefully adapt to the changes in the I/O access pattern, hence eliminating the need for explicit re-training decisions.

### 3.9 Summary

The high concurrency degree in modern applications makes recognizing higher level application access patterns challenging at the storage level, because the storage server sees random interleavings of accesses from different application streams. We introduce *QuickMine*, a novel caching and prefetching approach that exploits the knowledge of logical application sequences to improve prefetching effectiveness for storage systems.

*QuickMine* is based on a minimally intrusive method for capturing high-level application contexts, such as an application thread, database transaction, or query. *QuickMine* leverages these contexts at the storage cache through a dynamic, incremental approach to I/O block prefetching. We implement our context-aware, incremental mining technique at the storage cache in the Network Block Device (NBD), and we compare it with three state-of-the-art context-oblivious sequential and non-sequential prefetching algorithms. In our evaluation, we use three dynamic content applications accessing a MySQL database engine: the TPC-W e-commerce benchmark, the RUBiS auctions benchmark and DBT-2, a TPC-C-like benchmark. Our results show that context-awareness improves the effectiveness of block prefetching, which results in reduced cache miss rates by up to 60% and substantial reductions in storage access latencies by up to a factor of 2, for the read-intensive TPC-W and RUBiS. Due to the write intensive nature and rapidly changing access patterns in DBT-2, *QuickMine* has fewer opportunities for improvements in this benchmark. However, we show that our algorithm does not degrade performance by pruning useless prefetches for low performing contexts, hence avoiding unnecessary disk congestion, while gracefully adapting to the changing application pattern.
Chapter 4

Dynamic Resource Allocation

This chapter describes a novel multi-resource allocator to dynamically allocate resources for database servers using shared network-attached storage volumes.

4.1 Introduction

With the emerging trend towards server consolidation in large data centers, techniques for dynamic resource allocation for performance isolation between applications are becoming increasingly important. With server consolidation, operators multiplex several concurrent applications on each physical server of a server farm, connected to a shared network-attached storage. The benefits of server consolidation are reduced costs of management, power and cooling. However, multiplexed applications are in competition for system resources, such as, CPU, memory and disk, especially during load bursts. Moreover, in this shared environment, the system is still required to meet per-application performance goals. This gives rise to a complex resource allocation and control problem.

Currently, resource allocation to applications in state-of-the-art platforms occurs through different performance optimization loops, run independently at different levels of the software stack, such as, at the database server, operating system and storage server, in the consolidated storage environment shown in Figure 1.1. Each local controller typically optimizes its own local goals, e.g., hit-ratio, disk throughput, etc., oblivious to application-level goals. This might lead to situations where local, per-controller, resource allocation optima do not lead to the global optimum; indeed local goals may conflict with each other, or with the per-application goals [72]. Therefore, the main challenge in these modern enterprise environments is designing a strategy which adopts a holistic view of system resources; this
strategy should efficiently allocate all resources to applications, and enforce per-application quotas in order to meet overall optimization goals e.g., overall application performance or service provider revenue.

Unfortunately, the general problem of finding the globally optimum partitioning of all system resources, at all levels to a given set of applications is an NP-complete problem [82]. Complicating the problem are inter-dependencies between the various resources. For example, let us assume a two tier system composed of database servers and consolidated storage servers, and several applications running on each database server instance. For any given application, a particular cache quota setting in the buffer pool of the database system influences the number and type of accesses seen at the storage cache for that application. Partitioning the storage cache, in its turn, influences the access pattern seen at the disk. Hence, even deriving an off-line solution, assuming a stable set of applications, and available hardware e.g., through profiling, trial and error, etc., by the system administrator, is likely to be highly inaccurate, time consuming, or both. Due to these problems, with a few exceptions, previous work has eschewed dynamic resource partitioning policies, in favor of investigating mechanisms for enforcing performance isolation, under the assumption that per-application quotas, deadlines or priorities are predefined e.g., manually, for each given resource type [75, 115]. Examples of such mechanisms include CPU quota enforcement [10, 74], memory quota allocation based on priorities [13], or I/O quota enforcement between workloads [37, 49, 65].

We consider the problem of global resource allocation, which involves proportioning the database and storage server caches and the storage bandwidth among applications, according to overall performance goals. To achieve this, one needs to estimate the performance of an application, given different amounts of resources, i.e., the amount of memory and the disk bandwidth fraction. Given the resource-to-performance mapping for an application, we can then find the resource fractions to give to each application to meet performance goals.

We notice that there is a spectrum of methods to estimate the resource-to-performance mapping for an application. At one end, one can simply iterate over all configurations – that is, one can run the application on multiple hardware/software configurations and measure the application’s performance for each configuration. At the other end, we can build an analytical performance model that can predict the performance for all configurations. Other methods in the middle include techniques that predict the application’s performance using a simulation of the different components. There are trade-offs with each approach, which we explain next.
In a fully experimental sampling approach, the results are accurate i.e., the performance numbers are gathered from the system itself, but obtaining the results is time consuming. This delay comes from the fact that we need to actuate a new configuration and wait a sufficient time for the system to reach steady state to avoid any transient effects, e.g., waiting for the caches to warm-up. On the other hand, a performance model provides faster results but it may not be as accurate as obtaining results from the running system. The lower accuracy of a performance model is due to the approximations made in the model itself or in some cases the model is applicable for only certain application types, i.e., we can model a read-dominant application but not a write-dominant application.

An analytical performance model provides a fast approximation of the resource-to-performance mapping, and the experimentally gathered samples provide more accurate performance numbers for situations that cannot be described using a model. Therefore, our technique is to leverage the positives of each approach by building fast but approximate resource-to-performance mapping using one of the approximation approaches, and then incrementally refining the initial models by gathering experimental samples only for configurations where the initial models are inaccurate.

We focus on building a simple performance model in order to guide the search, by providing a good approximation of the overall solution. Our key ideas are to incorporate readily available information about the application and system into the performance model, and then refine the model through limited experimental sampling of actual behavior. Specifically, we reuse and extend on-line models for workload characterization, i.e., the miss-ratio curve (MRC) [115], as well as simplifications based on common assumptions about cache replacement policies. We further derive a disk latency model for a quanta-based disk I/O scheduler [103] and we parametrize the model with metrics collected from the on-line system, instead of using theoretical value distributions, thus avoiding the fundamental source of inaccuracy in classic analytical models [46]. The initial performance model provides a resource-to-performance mapping for each application, for many resource quota configurations. However, the initial model may only be an approximation or may not model the application in some configurations. To refine the model, we gather performance values by actuating the different configurations on the running system. We use statistical interpolation between computed and experimental sample points in order to re-approximate the per-application performance models, thus dynamically refining the model. We experimentally show that, by using this method, convergence towards near-optimal configurations can be achieved in minutes, while an exhaustive exploration of the multi-dimensional search space,
representing all possible partitioning configurations, would take weeks, or even months.

We implement our technique using commodity software and hardware components, without any modifications to interfaces between components, and with minimal instrumentation. We use the MySQL database engine running a set of standard benchmarks, i.e., the TPC-W e-commerce benchmark, and the TPC-C transaction processing benchmark. Our experimental testbed is a cluster of dual processor servers connected to a commodity storage hardware. We show experiments for on-line convergence to a global partitioning solution for sharing the database buffer pool, storage cache, and disk bandwidth in different application configurations. We compare our approach to two baseline approaches, which optimize either the memory partitioning, or the disk partitioning, as well as combinations of these approaches without global coordination. We show that for most application configurations, our computed model effectively prunes most of the search space, even without any additional tuning through experimental sampling. Our dynamic resource algorithm performs similar to an exhaustive search algorithm, but provides a solution within minutes, versus days/months of running time.

The remainder of this chapter is structured as follows. Section 4.2 provides a background on existing techniques for server consolidation in modern data centers, highlighting the need for a global resource allocation solution. We describe the resource allocation problem and present an overview of our solution in Section 4.3. We expand on our solution by describing each step in detail: building approximate performance models (Section 4.4), iteratively refining the model using runtime samples (Section 4.5), and finding the near-optimal configuration using the models (Section 4.6). Section 4.7 discusses how to handle general utility functions. Section 4.8 presents an example of our approach. Section 4.9 describes our prototype implementation. Section 4.10 presents the algorithms we use for comparison, and our experimental methodology. Section 4.11 presents the results of our experiments on this platform and Section 4.12 analyzes the accuracy of the models in depth. Section 4.13 concludes this chapter.

### 4.2 Motivation

In this section, we present results showing the lack of I/O resource allocation in current operating systems and highlight the need for multi-resource allocation.
4.2.1 Sharing Storage Bandwidth within an Operating System

In traditional servers, the data is stored on a local disk or on a directly-attached external storage device. In this design, the storage device simply reads and writes data corresponding to user I/O commands while the operating system on the server manages access to the storage device by scheduling the I/O requests. For example, in current operating systems such as Linux, the I/O scheduler implements request re-ordering to improve sequentiality and basic quality-of-service techniques to maintain fairness between processes sharing the disk bandwidth. We run simple experiments to show that the performance of an application can be severely affected when paired with another I/O intensive process, whether or not we enforce per-application CPU priorities at the operating system level. In particular, we show that the I/O scheduling policies implemented by the Linux operating system can run counter to application (or CPU) priorities leading to priority inversion reducing the performance of the high-priority application.

In our evaluation, we run two synthetic workloads: a small workload (Workload-A) with 1 outstanding I/O request and a large workload (Workload-B) with 10 outstanding I/O requests concurrently on the server with shared disk. The workloads are described in more detail in Section 4.10.2. The shared disk is a standard SATA disk that provides 75 IOPS for Workload-A and 91 IOPS for Workload-B when each workload is run alone on the server. Both Workload-A and Workload-B are generated using the ORION (Oracle IO Numbers) tool. We use the Linux operating system with either the cfq scheduler, that was recently added to the Linux kernel which attempts to provide fair queuing among several processes, and the traditional deadline scheduler in Linux that aims to minimize I/O seek time and prioritizes I/O reads over writes. The deadline scheduler does not support enforcement of CPU priorities. The cfq scheduler uses the process priorities (set using the nice utility) to favor the high-priority application.

Figure 4.1 shows the I/O interference between Workload-A and Workload-B when using the deadline scheduler with the Linux operating system. We see that, when Workload-A runs alone, it achieves 75 I/O operations per second (IOPS) and the average I/O latency 13.21 milliseconds. However, when Workload-B is co-scheduled with Workload-A, there is a significant slowdown. Workload-A's throughput is only 11% of its throughput running in isolation and the latency is a factor of 9 larger the original latency. In an attempt to achieve better performance for Workload-A, we set the CPU nice levels for Workload-A to -10 (high priority) and Workload-B to +10 (low priority) and re-run the experiment. With the deadline I/O
scheduler there is no difference after changing the process priorities.

The interference effect is lower when using the newer cfq scheduler in Linux. The cfq I/O scheduler tries to maintain fairness between processes sharing the disk. Thus, the performance of Workload-A is 44% of its throughput running in isolation and there is corresponding factor of 2.3 increase in its latency. This shows that while cfq I/O scheduler tries to maintain fairness, it does not fully achieve it. A fair share of resources would have led to Workload-A to 50% of its throughput obtained when running in isolation. Next, similar to the earlier experiments with the deadline I/O scheduler, we set the CPU nice levels for Workload-A to -10 (high priority) and set Workload-B to be +10 (low priority). The cfq uses the CPU nice levels to allocate more disk bandwidth to Workload-A improving its performance by 15% in terms of throughput.

The results show that while the cfq I/O scheduler performs better than the deadline scheduler, fine-grained control of the application’s performance is currently not provided by Linux. Furthermore, there is no method of communicating application SLO requirements and enforcing them at the storage server. Since both the OS and the storage server perform I/O scheduling in a per-application QoS oblivious manner, current architectures are unable to enforce end-to-end quality of service. As we have shown, this results in high performance degradation for the high priority application. In addition, we have shown that today’s operating systems, e.g., Linux, do not support the enforcement of application-level I/O requirements, leading to the degradation of performance of the high-priority application. While changes to the operating system’s I/O scheduling can improve this situation, we show next that better performance can only be achieved by looking at all resources involved in the I/O path such as caches and disk bandwidth.

### 4.2.2 Sharing Cache Space and Disk Bandwidth in a Storage Server

We present a simple motivating experiment that shows the need for multi-resource allocation. To simplify the presentation, we consider only accesses to the storage server, hence only the storage cache and the storage bandwidth resources. We run two synthetic workloads concurrently on the storage server: a small workload (Workload-A) with 1 outstanding request, and a large workload (Workload-B) with 10 outstanding requests, at any given time. Workload-A is cache friendly and achieves a cache hit ratio of 50% with a 1GB storage cache. In contrast, Workload-B is mostly un-cacheable; it obtains only a 5% hit ratio with a 1GB storage cache.

We run the workloads using several different configurations, i.e., uncontrolled sharing, partitioning the cache, disk or both between workloads. We normalize the latency of each
Figure 4.1: Running Workload-A/Workload-B with Deadline I/O Scheduler. We co-schedule Workload-A and Workload-B using the Linux Deadline I/O scheduler. As Workload-B is more intense than Workload-A, it *overwhelms* the I/O scheduler queue causing a large degradation in performance for Workload-A. The results also show that the Linux Deadline I/O scheduler does not consider process priorities in I/O scheduling where the performance of Workload-A does not improve despite it being marked as a high-priority process.

Figure 4.2: Running Workload-A/Workload-B with CFQ I/O Scheduler. We co-schedule Workload-A and Workload-B using the Linux CFQ I/O scheduler. The CFQ I/O scheduler tries to maintain fairness between processes thus the performance (in terms of throughput) is roughly half. However, the more intense Workload-B performs better than Workload-A. By changing the process priority, the performance of Workload-A improves but fine grained control is still needed.
workload relative to its latency running in isolation. Figure 4.3 presents our results. In all schemes, we use the combined application latencies (by simple summation) as the global optimization goal. We choose this simple metric for fairness of comparison with the miss-ratio curve algorithm [115], which optimizes the aggregate miss-ratio, hence the aggregate latency, while being agnostic to Service Level Objectives (SLOs) in general.

When running in isolation, Workload-A is able to utilize the 1 GB cache effectively and this results in an average storage access latency of 4.4ms. On the other hand, Workload-B does not benefit from the cache, resulting in an average storage access latency of 85.1ms. When the two workloads are run concurrently with uncontrolled resource sharing, the larger Workload-B dominates the smaller Workload-A at both cache and disk levels. This results in a factor of 6 slowdown for Workload-A and a factor of 4 slowdown for Workload-B. This result shows that workloads can suffer significant performance degradation when resource sharing is not controlled.

Next, we run the workloads using different resource partitioning algorithms. First, we partition the storage cache using the miss-ratio curves of the workloads [115], while disk bandwidth sharing is uncontrolled. The MRC algorithm determines that the best cache setting is to allocate the bulk of the storage cache (992 MB) to Workload-A and provide a minimum to Workload-B. Cache partitioning thus improves the performance of Workload-A.
significantly from 26.6ms to 19.9ms. Next, we iterate through all possible disk partitioning settings to find the best disk bandwidth partitioning between the workloads, and enforce it using quanta-based scheduling [103], while cache sharing is uncontrolled. By partitioning the disk bandwidth, the performance of Workload-A improves to 13.2ms. In addition, Workload-B improves to 169.7ms. While properly partitioning the resource at each level independently, as described above, alleviates the interference, neither partitioning results in the optimal configuration for these two workloads.

On the other hand, an exhaustive search of both the cache and bandwidth settings yields an ideal setting where the storage access latency is 9.64ms for Workload-A and 171.3ms for Workload-B. In our simple case, the allocation solution found by the exhaustive search algorithm is just a combination of the solutions found by the two independent partitioners, for cache and disk. However, as we will show, due to the interdependence between resources, this is not the case when more resources are considered. Finally, iterating through all possible configurations and taking experimental samples for the exhaustive search is clearly infeasible for non-trivial combinations of resources and workloads.

These experiments and observations thus motivate us to design and implement a coordinated multi-resource partitioning algorithm based on an approximate system and application model, which we introduce next.

4.3 Problem Statement and Overview of Solution

In this section, we describe our approach to provide effective resource partitioning for database servers running on virtual storage. Our main objective is to meet an overall performance goal, e.g., to minimize the overall latency, when running a set of database applications on a shared storage server. In order to achieve this, we use the following:

1. A performance model based on minimal statistics collection in order to approximate a resource-to-performance mapping,

2. An iterative statistical interpolation technique that refines the initial model and,

3. A resource allocation algorithm that uses the per-application performance models to find a near-optimal partitioning of resource among several applications.

In the following, we first introduce the problem statement, and an overview of our approach. Then, we introduce our performance model, and its sampling-based fine-tuning in detail.
4.3.1 Problem Statement

We study dynamic resource allocation to multiple applications in dynamic content servers with shared storage. In the most general case, let us assume that the system contains $m$ resources and is hosting $n$ applications. Our goal is to find the optimal configuration for partitioning the $m$ resources among the $n$ applications. Let us denote with $R_1, R_2, \ldots, R_n$ the data access times of the $n$ applications hosted by the service provider. For the purposes of this dissertation, we assume that the goal of the service provider is to minimize the sum of all data access latencies for all applications, i.e.,

$$U = \min \sum_{i=1}^{n} R_i \quad (4.1)$$

However, our approach does not depend on the particular goal we set. For example, alternatively, we can optimize the provider’s revenue expressed as a utility function based on the application latencies. We describe how our approach can be applied to general utility functions in Section 4.7. Regardless of the goal we set, we assume that our algorithm is aware of that goal, and can monitor application performance in order to compute the total benefit obtained for all applications, in any resource quota configuration. Finding a practical solution to this problem is difficult, because the optimal resource allocation depends on many factors, including the (dynamic) access patterns of the applications, and how the inner mechanisms of each system component e.g., cache replacement policies, affect the inter-dependencies between system resources.

4.3.2 Overview of Solution

Our technique determines per-application resource quotas in the database and storage caches, on the fly, in a transparent manner, with minimal changes to the DBMS, and no changes to existing interfaces between components. Towards this objective, we use an online performance estimation algorithm to dynamically determine the mapping between any given resource configuration setting and the corresponding application latency. While designing and implementing a performance model for guiding the resource partitioning search is non-trivial, our key insight is to design a model with sufficient expressiveness to incorporate (i) tracking of dynamic access patterns, and (ii) sufficiently generic assumptions about the inner mechanisms of the system components and the system as a whole.
For this purpose we collect a trace of I/O accesses at the DBMS buffer pool level and we use periodic sampling of the average disk latency for each application in a baseline configuration, where the application is given all the disk bandwidth. We feed the access trace and baseline disk latency for each application into our algorithm, which computes the latency estimates for that application for all possible resource configurations. We thus obtain a set of resource-to-performance mapping functions, i.e., *performance models*, one for each application. Next, we enhance the accuracy of each performance model through experimental sampling. We use statistical regression to re-approximate the performance model by interpolating between the pre-computed and experimentally gathered sample points. We then use the corresponding per-application performance models to determine the near-optimal allocation of resources to applications according to our overall goal. Specifically, we leverage the derived performance model of each application, and use *hill climbing* [85] to converge towards a partitioning setting that minimizes the combined application latencies. In the following sections, we describe our model that estimates the performance of an application using multi-level caches and a shared disk.

4.4 Building Approximate Performance Models

We use three key insights about the inner workings of the system, as explained next, to derive a close performance approximation, while at the same time reducing the complexity of the model as much as possible.

4.4.1 Key Assumptions and Ideas

The key assumptions we use about the system are (i) that the database applications are I/O intensive, (ii) that the cache replacement policy used in the cache hierarchy is known to be either the standard, uncoordinated LRU, or the coordinated DEMOTE [111] policy and (iii) that the server is a closed-loop system i.e., it is interactive and the number of users is constant during periods of stable load. These assumptions match our target system well, leading to a performance model with sufficient accuracy to find a near-optimal solution, as we will show in Section 4.11. With the assumptions above, our key idea is to reduce the search space of configurations with insights of the system. For example, if the cache replacement policies follow LRU then the performance of a two-level cache hierarchy can be modeled using the performance of a single level of cache. Our insights into the system allow us to obtain a close performance estimation, at higher speed, as described next.
We provide a list of the key symbols used in the derivation of the approximate performance model.

We approximate the cache hierarchy with the model of a single-level cache, and we specialize this model for two most commonly deployed, or proposed cache replacement policies, i.e., uncoordinated LRU and coordinated DEMOTE [111]. We also derive a disk model that predicts the application’s I/O latency given different fractions of the disk bandwidth. Based on our models, assuming that the application is given quotas i.e., fractions $\rho_c$, $\rho_s$ and $\rho_d$ of the buffer pool cache, storage cache and disk bandwidth, respectively, we estimate the overall data access latencies for the respective quotas through a combination of selective on-line measurements and computation.

In the following, we first introduce an approximation of the cache miss-ratio of a two-level cache hierarchy, $\hat{M}(\rho_c, \rho_s)$, as a function of the cache quotas $\rho_c$ and $\rho_s$, for the two types of replacement policies we consider. Then we introduce our disk model that computes the disk latency as a function of the disk quota, $L_d(\rho_d)$. Finally, we describe our overall data access latency model.

### 4.4.2 Modeling the Performance of a Two-Level Cache Hierarchy

In a cache hierarchy using the standard (uncoordinated) LRU replacement policy at all levels, any cache miss from cache level $q_i$ will result in bringing the needed block into all lower levels of the cache hierarchy, before providing the requested block to cache $i$. It follows that the block is redundantly cached at all cache levels, which is called the *inclusiveness*
property [111]. Therefore, if an application is given a certain cache quota \( q_i \) at a level of cache \( i \), any cache quotas \( q_j \) given at any lower level of cache \( j \), with \( q_j < q_i \) will be mostly wasteful.

In contrast, in a cache hierarchy using coordinated DEMOTE cache replacement, when a block is fetched from disk, it is not kept in any lower cache levels. The lower cache levels cache blocks only when the block is evicted from a higher cache level. Therefore, the application benefits from the combined quotas at all levels due to cache exclusiveness [111]. Based on these observations, we make the following simplifications to approximate the overall miss-ratio of a two-level cache, i.e., \( \widehat{\mathcal{M}}(\rho_c, \rho_s) \), based on a single-level cache model.

In an uncoordinated LRU cache hierarchy, only the maximum size quota given at any level of cache matters; therefore, we approximate the miss-ratio of a two level cache, consisting of a buffer pool (with quota \( \rho_c \)) and a storage cache (with quota \( \rho_s \)) by the following formula

\[
\widehat{\mathcal{M}}(\rho_c, \rho_s) \approx \mathcal{M}_c(\max[\rho_c, \rho_s])
\] (4.2)

In a coordinated DEMOTE cache hierarchy, the combined cache quotas given to the application at all levels of cache has the same effect on the overall miss-ratio as giving the total quota in a single level of cache. Therefore, for DEMOTE cache replacement, we use the following formula to approximate the miss-ratio of a two-level cache

\[
\widehat{\mathcal{M}}(\rho_c, \rho_s) \approx \mathcal{M}_c(\rho_c + \rho_s)
\] (4.3)

We can further approximate the fraction of accesses that miss in both levels of cache, hence reach the disk, i.e., \( \mathcal{M}_c(\rho_c)\mathcal{M}_s(\rho_c, \rho_s) \) from the formula above, with the fraction of disk accesses given by the miss-ratio of our previously introduced single-level cache model as

\[
\mathcal{M}_c(\rho_c)\mathcal{M}_s(\rho_c, \rho_s) = \widehat{\mathcal{M}}(\rho_c, \rho_s)
\] (4.4)

By using the previously derived models for \( \widehat{\mathcal{M}}(\rho_c, \rho_s) \) e.g., in the case of uncoordinated
LRU (Equation 4.2), we obtain

\[
\mathcal{M}_s(\rho_c, \rho_s) = \frac{\mathcal{M}_c(\max[\rho_c, \rho_s])}{\mathcal{M}_c(\rho_c)}
\]  \hspace{1cm} (4.5)

Therefore, we can approximate the miss-ratio in the storage cache, \(\mathcal{M}_s(\rho_c, \rho_s)\), in terms of the miss-ratio of a single-level cache model.

### 4.4.3 Modeling the Disk Performance

Our storage system uses the quanta-based scheduler to divide the storage bandwidth among several virtual volumes. The quanta-based scheduler partitions the bandwidth by allocating a time quantum where one of the workload obtains exclusive access to the underlying disk. For modeling the disk latency, we observe that the typical server system is an interactive, closed-loop system. This means that, even if incoming load may vary over time, at any given point in time, the rate of serviced requests is roughly equal to the incoming request rate. According to the interactive response time law [46],

\[
L_d = \frac{N}{X} - z
\]  \hspace{1cm} (4.6)

where \(L_d\) is the response time of the storage server, including both I/O request scheduling and the disk access latency, \(N\) is the number of application threads, \(X\) is the throughput, and \(z\) is the think time of each application thread issuing requests to the disk.

We then use this formula to derive the average disk access latency for each application, when given a certain quota of the disk bandwidth. We assume that think time per thread is negligible compared to request processing time, i.e., we assume that I/O requests are arriving relatively frequently, and disk access time is significant. If this is not the case, the I/O component of a workload is likely not going to impact overall application performance. However, if necessary, more precision can be easily afforded e.g., by a context tracking approach, which allows the storage server to distinguish requests from different application threads [99], hence infer the average think time. We further observe that the throughput of an application varies proportionally to the fraction of disk bandwidth given to the application. Since disk saturation is unlikely in interactive environments with a limited number of I/O threads, this is very intuitive, but also verified through extensive validation experiments using a quanta-based scheduler and a variety of workloads.
Through a simple derivation, we arrive at

\[ L_d(\rho_d) = \frac{L_d(1)}{\rho_d} \]  

(4.7)

where \( L_d(1) \) is the baseline disk latency for an application, when the entire disk bandwidth is allocated to that application. This formula is intuitive. For example, if the entire disk was given to the application, i.e., \( \rho_d = 1 \), then the storage access latency is equal to the underlying disk access latency. On the other hand, if the application is given a small fraction of the disk bandwidth, i.e, \( \rho_d \approx 0 \), then the storage access latency is very high (approaches \( \infty \)).

Finally, the total cache quota allocated to an application influences the arrival rate of I/O requests at the disk, hence the baseline disk latency for that application. For example, a larger cache quota may result in a smaller disk queue, which in its turn limits opportunities for scheduling optimizations to minimize disk seeks. Hence, in the absence of disk bandwidth saturation, a larger cache quota may result in a higher baseline disk latency for the corresponding application. Therefore, to compute the baseline disk latency for an application given a particular cache configuration, we use linear interpolation based on experimental measurements, taken for a few cache settings, instead of a single measurement.

4.4.4 Computing the Overall Performance Model

Assuming that the hit access latency in the buffer pool is negligible, the overall latency is determined by the accesses that miss in the buffer pool and either (i) hit in the storage cache or (ii) miss in the storage cache and hence access the disk. We also assume that the access latency for a hit/miss in the storage cache is approximately the network/disk latency, i.e., \( L_{net}/L_d \), respectively, then the average application latency is

\[
R_{\text{avg}}(\rho_c, \rho_s, \rho_d) = \frac{M_c(\rho_c)H_s(\rho_c, \rho_s)L_{\text{net}}}{\text{I/Os satisfied by the storage cache}} + \frac{M_c(\rho_c)M_s(\rho_c, \rho_s)L_d(\rho_d)}{\text{I/Os satisfied by the disk}}
\]

(4.8)

where the miss (and hit) ratio at the storage cache, \( M_s(\rho_c, \rho_s) \), is a function of both the quota at the first level cache (\( \rho_c \)), and the quota at the second level cache (\( \rho_s \)), while the miss-ratio of the buffer pool, \( M_c(\rho_c) \), is only a function of \( \rho_c \). We can further approximate the fraction of accesses that miss in both levels of cache, hence reach the disk, \( M_c(\rho_c)M_s(\rho_c, \rho_s) \), from
the formula above, with the fraction of disk accesses given by the miss-ratio of our previously introduced single-level cache model as

\[ M_c(\rho_c)M_s(\rho_c, \rho_s) = \hat{M}(\rho_c, \rho_s) \]  

(4.9)

By using the previously derived models for \(\hat{M}(\rho_c, \rho_s)\) e.g., in the case of uncoordinated LRU (Equation 4.2), we obtain

\[ M_s(\rho_c, \rho_s) = \frac{M_c(\max[\rho_c, \rho_s])}{M_c(\rho_c)} \]  

(4.10)

Therefore, we can approximate the miss-ratio in the storage cache, \(M_s(\rho_c, \rho_s)\), in terms of the miss-ratio of a single-level cache model. By replacing the respective miss/hit ratio of the storage cache in Equation 4.8, we derive the application latency based on our single-level cache performance model for either type of cache replacement policy.

Finally, in order to derive a complete resource-to-performance model, we perform access trace collection and compute the miss-ratio curve (MRC) only at the buffer pool level. Then, we vary the quota allocations for the two caches and the disk bandwidth for the application, to all possible combinations in the model. For each quota setting, we then compute the corresponding application latencies, based on the precomputed buffer pool MRC, by Equation 4.8.

4.5 Runtime Refinement of Performance Models

A multi-tier storage system is a complex system consisting of many hardware and software components. Therefore, it is difficult to model all aspects of the system. In our approach, we build lightweight approximations of the components critical to the performance of the application, i.e., the sizes of the database and storage caches, and the fraction of the disk bandwidth allocated to each application. However, the true model of an application can only be obtained by experimentally actuating all configurations. The result of experimentally running each configuration provides us with an accurate measurement of the application’s performance but obtaining all samples is time consuming often requiring several months to
complete. Every other method (including our own) makes some simplifications to arrive at a near-perfect approximation to the true model.

In general, there is a spectrum of approaches to derive a performance model of an application. It ranges from an analytical model, which can be derived mathematically, to a black-box model, where a model is generated by gathering performance samples from the system, using different configurations. There are trade-offs with each approach. At one end, deriving an analytical model is time-consuming and it requires a detailed understanding of the underlying system. In addition, for highly complex workloads, deriving an analytical model may not be feasible. At the other end is a black-box model where one actuates the system with different configurations and gathers performance samples – these performance values can then be interpolated using statistical regression techniques to build the overall model. Our approach is to leverage the positives of each approach – that is, we use an analytical model to predict the performance of applications and components with well-understood behavior (e.g., we assume that the cache replacement policy is based on LRU) and refine the models with performance samples obtained at runtime.

In our simple performance model we ignore the effects of locking for concurrency control, dirty block flushes for the cache model, actions taken by the DBMS recovery manager (i.e., the write-ahead logging), and imperfect I/O isolation at small disk quotas for the disk model. Specifically, whenever a dirty block evicted from the buffer pool is flushed to disk, the write access goes through all lower levels of cache on its way out. Hence, the evicted block remains cached in the storage cache, violating our assumption of redundancy for uncoordinated LRU caches, hence impacting cache miss-ratio predictions. We describe the areas of inaccuracy of our performance models and the method of correcting these models next.

4.5.1 Limitations of the Cache Model

Our analytical cache model (described in Section 4.4.2) assumes the cache replacement policies to be LRU (or closely approximate LRU), assumes the workload to be mostly reads, and ignores locking for concurrency control. Despite these shortcomings, the analytical model approximates the behavior of a two-level cache hierarchy well. As we show in Section 4.12.1, the primary areas of error are for configurations where the two-level caches are roughly equal and when the (incoming) workload contains more than 10% of writes.

The two-level analytical cache model does not capture the complex behaviors created by writes in cache accesses as this creates accesses to the second-level cache on first-level cache evictions from block flushes (i.e., writing back a dirty block). We show the different actions
Figure 4.4: **Cache behavior.** A cache access can be handled in 3 ways. First, it can result in a *cache hit*, where the data is found in the cache. Second, it can result in a *cache miss* where the data is brought into the cache from the disk. Finally, it can result in a *cache flush* and a *cache miss* where a previously cached block is written back to disk before the new data block is brought into the cache.

An access to a cache results in one of three actions: (i) *cache hit* – where the item is found in the cache, (ii) *cache miss* – where the item is not found in cache and needs to be fetched from the underlying disk, and (iii) *cache flush and cache miss* – where we have to flush an entry from the cache (i.e., writing back a dirty block) then issue a disk read to read in the item. We only have the first two behaviors in a read-only workload leading us to accurately model the performance of the cache as

\[
L_{avg}(\rho_c) = \frac{H_c(\rho_c)L_{hit}}{Cache \ hit} + \frac{M_c(\rho_c)L_{disk}}{Cache \ miss}
\]  

(4.11)

However, when the workload has writes as well, the above model fails to account for cache flushes leading to poor performance estimation. Specifically, the model for a single-level
cache is corrected as

\[
L_{\text{avg}}(\rho_c) = \frac{H_c(\rho_c) L_{\text{hit}}}{L_{\text{hit}}} + \frac{M_c(\rho_c) L_{\text{disk}}}{L_{\text{disk}}} + \gamma L_{\text{disk}} \]

where \(\gamma\) is the fraction of cache misses that result in a cache flush. For a read-only workload, \(\gamma = 0\), as no blocks in the cache are dirty, and \(\gamma = 1\) for a write-only workload where every cache miss results in a cache evict to writethback the dirty block. In general, it is difficult to predict the parameter \(\gamma\), i.e., the fraction of cache evictions that result in a cache flush. A naïve solution is to consider the write fraction of the workload to determine the fraction of cache misses resulting in a block flush. However, this is not a good indicator of \(\gamma\), as we explain next. Consider a workload where the blocks are accessed uniformly on disk, resulting in a straight line for the miss-ratio curve. In this workload, 25\% of the blocks may be written, leading to a write ratio of 25\% or all of the blocks may be written 25\% of the time also leading to a 25\% write ratio. While both have the same write ratio, the resulting miss trace is different; the workload where 25\% of the blocks are written has fewer block flushes than the workload where the blocks are written 25\% of the time.

The extension of the full cache model i.e., by accounting for cache flushes, to a two-level cache model is more complex. The three actions taken by the first-level cache correspondingly turn into three actions at the second-level cache and the combinations lead to twelve different outcomes for a cache access. These include three choices on a first-level cache miss and nine outcomes for a first-level cache flush and cache miss. Modeling all interactions between the two-levels of caches is difficult. To correct the initial predictions, we can model the behavior of two-levels of cache using cache simulation where we replay the access trace and record actions taken at each level to handle the cache access. This approach works well and captures all possible outcomes. However, DBMS systems, such as MySQL/InnoDB, implement specialized cache flushing algorithms that deviate from the standard LRU replacement algorithms. For instance, MySQL/InnoDB flushes (spatially) neighboring dirty blocks when flushing a dirty block from the end of the LRU list to optimize for disk performance. These neighboring blocks may be more recent (higher on the LRU stack) thereby the optimization violates the LRU replacement policy. Such optimizations could be incorporated into the simulator for an open-source DBMS but is not possible for closed-source DBMS. Instead, to have our methods be applicable to many systems, we choose to correct the errors in the cache model by augmenting the model with samples gathered experimentally at runtime.
4.5.2 Limitations of the Disk Model

In our approximate performance model of the disk, we make the assumption that the disk is highly utilized – that is, there are many users in the system or the think-time between requests is small. In a closed-loop system with high utilization, each of the \( N \) requests in the disk scheduler queue must wait for \( N - 1 \) clients to exit the system before obtaining a chance to run on disk leading the throughput of one request to be \( \frac{1}{ND+Z} \) and the cumulative throughput for \( N \) requests is

\[
X = \frac{N}{ND + Z} \quad (4.13)
\]

Given our assumptions that (i) the think-time is small \((Z \to 0)\) or, (ii) the number of clients is large \((N \to \infty)\), we find that the throughput \( X \) approaches \( \frac{1}{D} \).

A second source of error in the disk model is due to the underlying disk firmware – that is, the optimizations and scheduling decisions taken by the disk firmware is affected by the traces of I/O requests issued to the disk. Moreover, for low disk quanta, the disk scheduler incurs frequent and potentially large disk seeks between the data locations of different applications on disk. Thereby, our disk latency prediction, as well as the underlying I/O bandwidth isolation mechanism itself would be inaccurate in this case. In particular, the disk quanta cannot be less than the maximum duration of a disk read/write, which is that of a block size of 16KB in our case (for MySQL).

4.5.3 Iterative Refinement of Performance Models

In order to fine-tune our performance model at run time, and iteratively correct any inaccuracies, we use a sampling-based approach to correct the model at runtime. We collect experimental samples of application latency in various resource partitioning configurations, and use statistical regression i.e., support vector machine regression [29] (SVR), to re-approximate the resource-to-performance mapping function without sampling the search space exhaustively. SVR allows us to estimate the performance for configuration settings we have not actuated, through interpolation between a given set of sample points.

SVR is a nonlinear regression algorithm that scales well for highly-dimensional and nonlinear data. It works as follows. Given a set of training points

\[
S = \{(\vec{x}_1, y_1), \ldots, (\vec{x}_m, y_m)\}
\]
Algorithm 1 Finding the Optimal Resource Partitioning Settings $C^*$

1: **Initialize:** $\forall i$, sample set $S_i$ of application $i$, $S_i = \emptyset$
2: /* Build approximate performance models */
3: for $i = 1$ to $n$ do
4: 1) Obtain initial performance samples and traces
5: 2) Build approximate performance model, $R_i^{\text{model}}$
6: 3) Get performance estimates from model, $S_i = S_i$ from model $R_i$
7: end for
8: /* Iteratively refine the initial model */
9: while any model ($R_i$) needs refining or performance ($U$) is still improving do
10: for $i = 1$ to $n$ do
11: if $R_i$ needs to be refined then
12: 1) Add new samples to sample set $S_i = S_i \cup \{(\vec{x}, y)\}$ (replace old samples)
13: 2) Use SVR to learn the function $R_i$ using sample set $S_i$
14: 3) Evaluate goodness of new model
15: end if
16: end for
17: 4) Find $U = \min \sum_{i=1}^{n} R_i(c_i)$ for all valid configurations.
18: end while
19: **Output:** Current best configuration $C^*$ which generates best $U$.

SVR finds a smooth function $\hat{f}(\vec{x})$ that has a small deviation ($\epsilon$) from the targets $y_i$ for all training data. The estimated function $\hat{f}(\vec{x})$ takes the form

$$\hat{f}(\vec{x}) = \sum_{i=1}^{m} \alpha_i y_i K(\vec{x}_i, \vec{x})$$

(4.14)

where each training point $\vec{x}_i$ is associated with a variable $\alpha_i$ that represents the strength with which the training point is embedded in the final function. The points which lie closest to the hyperplane, denoting $\hat{f}(\vec{x})$, are called the support vectors. $K(\vec{x}_i, \vec{x})$ is a kernel function which maps the input into a high dimensional space, called feature space, where linear support vector regression is applied. We use radial basis functions (RBFs) as our kernel functions. Once the data is transformed using the kernel functions, the training of SVR consists of solving a convex optimization problem using quadratic programming.

Our iterative algorithm approximates a performance hyperplane e.g., surface, per application, based on a set of sample points that it expands incrementally in each iteration. Each sample point represents the measured average data access latency of the application, given a specific resource partitioning configuration $c_i$. In each iteration, we add additional sample points to the SVR training sample set, through measurements obtained from experiments.
to increase the accuracy of the regression function, and thus speed up convergence of the SVR algorithm.

Algorithm 1 shows the pseudo-code of our iterative refinement process. We iteratively collect a set of $k$ (16 or 32) randomly chosen sample points; each sample represents the average application latency measured in a given configuration. We explain the rationale of choosing 16 or 32 samples in Section 4.12.4. We replace the respective points in our performance model with the new set of experimentally collected samples (as these samples are more accurate). Then, using all sample points, consisting of both computed and experimentally collected samples, we retrain the regression model. Specifically, at a given iteration of the algorithm, each application $i$ has a sample set, denoted by $S_i$, initialized to empty (line 1). We first create approximate performance models using a small number of performance samples and traces collected from the running system (lines 2-7). Then, in each iteration step, for each application $i$, we generate a new set of sample points to expand the current sample set (line 12); we then learn the regression functions $R_i(c_i)$ based on the current sample set. We stop the refinement process (lines 8-18) when all models have been refined or if the overall performance has stabilized across iterations.

The learning algorithm converges when either one of the following conditions occurs: (i) adding more sampling points does not increase the accuracy of the regression function i.e., the per-application hyperplane representations vary only within a predefined deviation bound across iterations, or (ii) the sum of performance of all applications does not change anymore across iterations, even with increasing the resolution of the regression functions. In the first case, we use cross-validation; we train the regression model on a subset of the samples and compare the resulting regression function using the remaining samples. If during cross-validation, we determine that the regression-based performance model is stable [29], then we conclude that we do not need to collect any more samples, and we have achieved a highly accurate performance model for the respective application. In the second case, we use a hill climbing approach to find the best performance corresponding to the hyperplane approximations for all applications. If the performance, i.e., the sum of application latencies, does not improve across iterations – indicating that the addition of performance samples is not improving the overall performance goal – we state that the performance model has converged. Hence, the more accurate the initial performance model, the lower the amount of sampling is needed to fine-tune the model. Conversely, if the initial performance model is highly inaccurate e.g., because completely different cache replacement policies than built into the performance model are deployed, then exhaustive sampling will be eventually performed.
to replace all initial model-generated sample points.

4.6 Finding the Optimal Configuration

Based on the per-application performance models derived as above, we find the resource partitioning setting which gives the optimum i.e., lowest combined latency in our case, by using hill climbing with random-restarts [85]. The hill climbing (conversely known as gradient descent) algorithm is an iterative search algorithm that moves towards the direction of increasing combined utility value for all valid configurations at each iteration. To avoid reaching a local optimum, we conduct several searches from several points chosen randomly until each search reaches an optimum. We use the best result obtained from all searches.

4.7 Supporting General Utility Functions

We assume our optimization goal is to minimize the sum of application latencies. We choose this goal to be independent of any bias created by an utility function that may change the configuration towards improving revenue for the datacenter provider rather than improving performance of all applications. However, we note that datacenter providers often do associate a different utility for each application. The utility function corresponding to the performance of any given application may vary depending on the contract between the service provider and client, the costs of the provider when hosting the application, etc. Our performance models do not depend on the exact specification of the utility function. For completeness, we also explore the effect of utility functions on resource allocation by defining utility functions as follows.

We classify applications into two categories: high-priority applications and best-effort applications. Figure 4.5a depicts the utility function we use for high-priority applications. For this application class, the provider pays a penalty whenever the application’s SLO i.e., its average data access latency (denoted as response time in the figure), is violated beyond a small margin of error called slack. On the other hand, the provider has no benefits for providing service better than the pre-agreed SLO for the application. We believe this scenario exists when a customer agrees to a long-term contract with the service provider. As shown in the figure, as long as the application’s response time is less than a deadline $D$ with some slack $\epsilon$, the utility is constant at zero. Beyond this value, the provider starts paying penalties for SLO violations, proportional to the magnitude of the violation, until
another threshold $D'$ considered to be unacceptable to the customer.

Figure 4.5b shows the utility function for the best effort application class. The provider pays no penalties, regardless of the level of service for an application in this class. Hence, the baseline level of performance with response time beyond $D'$ has the utility value zero. For example, this baseline level would correspond to the application performance for 100% cache miss-ratio for any level of cache in our case. However, we assume that performance above the baseline carries a reward for the service provider, which increases proportionally to the level of service until the performance level that would correspond to a maximum performance level, after which no more benefits accrue.

4.8 Analytical Derivation of a Sample Scenario

In this section, we analytically find the optimal resource partitioning setting for two applications sharing two levels of uncoordinated LRU caches and the disk bandwidth. For simplicity, we assume that the applications are identical and issue only reads. This implies that the two applications (i) have the same miss-ratio curves, (ii) have the same disk performance ($L_{disk}$) and, (iii) have the same network performance ($L_{net}$). We further assume that the applications are accessing the disk volume uniformly (but randomly) a working set of 1GB leading to a miss-ratio curve described by

$$M(\delta) = 1 - \frac{\delta}{1024} \tag{4.15}$$

where $\delta$ is the fraction of the cache (in MB) given to the application. The curve states that the miss-ratio is 1.0 (all misses) when the application is given no cache and 0.0 (all hits) when given a cache equal to the size of the underlying disk volume (which is 1GB). We make this assumption of the data access pattern to allow us to simplify our derivation but it may be replaced with a different miss-ratio curve without loss of generality. Also, for mathematical brevity, we can re-write the above equation as

$$M(\rho) = 1 - \rho \tag{4.16}$$

where $\rho = \frac{\delta}{1024}$. Then, using our multi-level cache approximation, we obtain
Figure 4.5: **Utility Functions.** We show the utility functions for two application classes: *high-priority* and *best-effort*. We assume that a violation of the SLO for *high-priority* application results in a penalty for the datacenter provider. The *best-effort* application provides revenue proportional its performance – where good performance yields more revenue and poor performance yields little or no revenue but the datacenter is not penalized for the poor performance of the *best-effort* application.
R(ρc, ρs, ρd) = \begin{cases} ρcL_{hit} + (1 - ρc) \frac{L_{disk}}{ρd} & ρc \geq ρs \\ ρcL_{hit} + (1 - ρc) [1 - \frac{1 - ρs}{1 - ρc}] L_{net} + (1 - ρc) \frac{1 - ρs}{1 - ρc} \frac{L_{disk}}{ρd} & ρc < ρs \end{cases} \tag{4.17}

as the latency of one application given resources ρc in the first level cache, ρs at the second level cache, and ρd as the disk bandwidth fraction.

The utility for the service provider, running these two applications α and β is

\[ U = R(ρ^α_c, ρ^α_s, ρ^α_d) + R(ρ^β_c, ρ^β_s, ρ^β_d) \tag{4.18} \]

where \( R(ρ_c, ρ_s, ρ_d) \) is the latency of one application as defined in Equation 4.17.

As the resources are only shared between two applications, the above equation can be re-written as

\[ U = R(ρ^α_c, ρ^α_s, ρ^β_d) + R(1 - ρ^α_c, 1 - ρ^α_s, 1 - ρ^β_d) \tag{4.19} \]

allowing us to reduce the number of parameters. Specifically, it allows us to write Equation 4.17 as

\[ U = \underbrace{ρ^α_c L_{hit} + (1 - ρ^α_c) (1 - \frac{1 - ρ^α_s}{1 - ρ^α_c}) L_{net} + (1 - ρ^α_c) \frac{1 - ρ^α_s}{1 - ρ^α_c} \frac{L_{disk}}{ρ^α_d}}_{\text{Application } α} + \underbrace{ρ^β_c L_{hit} + (1 - ρ^β_c) \frac{L_{disk}}{ρ^β_d}}_{\text{Application } β} \tag{4.20} \]

since if ρ^α_c < ρ^α_s then ρ^β_c ≥ ρ^β_s. After substituting the variables for Application β (as in Equation 4.19), we get

\[ U = \frac{(1 - ρ^α_c) L_{disk}}{ρ^α_d} + \frac{ρ^α_c L_{disk}}{1 - ρ^α_d} + (ρ^α_s - ρ^α_c) L_{net} + L_{hit} \tag{4.21} \]

where parameters 0 ≤ ρ^α_c ≤ 1, 0 ≤ ρ^α_s ≤ 1, and 0 ≤ ρ^β_d ≤ 1. In addition, we started with the assumption that ρ^α_c < ρ^α_s and from the specifications of the underlying hardware, we know that \( L_{disk} \gg L_{net} \gg L_{hit} \geq 0 \). Thus, we can minimize \( U \) by eliminating the first
two terms of Equation 4.21 by setting $\rho_s^\alpha = 1$ and $\rho_c^\alpha = 0$ leading to

$$
U = \frac{L_{hit}}{\text{Application } \alpha} + \frac{L_{net}}{\text{Application } \beta}
$$

(4.22)

as a valid solution. This shows that the configuration that minimizes the overall latency, $U$, is when Application-$\alpha$ is given none of the first-level cache and all of the second-level cache. Conversely, it can be though of as Application-$\beta$ is given all of the first-level cache and none of the second-level cache. The resulting configuration is intuitive as the average latency of Application-$\beta$ is $L_{hit}$ since all data accesses hit the first-level cache and the average latency of Application-$\alpha$ is $L_{net}$ as all accesses hit in the second-level cache.

### 4.9 Prototype Implementation

We extend our infrastructure (described in Chapter 3) to implement dynamic resource allocation. Specifically, we add a resource controller in charge of partitioning multiple levels of storage cache hierarchy and the storage bandwidth. The controller determines per-application resource quotas on the fly, based on our performance models, in a transparent manner, with minimal changes to the DBMS i.e., to collect access traces at the level of the buffer pool and to monitor performance.

We also modify the MySQL/InnoDB buffer pool implementation to support dynamic partitioning and resizing of the buffer pool for each workload partition, as MySQL does not currently provide these features. We limit the number of buffer pool pages an application may use and once its quota is fully used, new blocks are placed into spaces created by evicting old blocks from the application’s quota. Specifically, in our design, each workload is given a limit of how many buffer pool pages it may use. It obtains buffer pool pages from a common pool until the limit is met and it is prevented from obtaining past its buffer pool limit. Once its quota is fully used, new page requests are satisfied by evicting an old page from the workload’s partition.

### 4.10 Evaluation

In this section, we describe several resource partitioning algorithms we use in our evaluation. In addition, we describe the benchmarks and methodology we use.
4.10.1 Algorithms used for Comparison

We compare our GLOBAL+ resource partitioning scheme, where we combine performance estimation and experimental sampling, with the following resource partitioning schemes.

1. **GLOBAL**: Is our resource allocation scheme where we use only the performance model. As opposed to the GLOBAL+ scheme; we do not add any runtime performance samples.

2. **MRC**: Uses MRC to perform cache partitioning *independently* at the buffer pool and the storage cache, based on access traces seen at that level. The disk bandwidth is equally divided among all applications.

3. **DISK**: Assigns equal portions of the cache to all applications at each level and explores all the possible bandwidth configurations at the disk level.

4. **MRC+DISK**: Uses the cache configurations produced by the MRC scheme and then experimentally explores all the possible configurations for partitioning the disk bandwidth.

5. **IDEAL**: Finds the configuration with best overall latency by an exhaustive search through all possible cache and disk partitioning configurations. For a thorough evaluation, we ran every configuration for 1 hour for all three workloads over a period of 11 months (between Dec/08 and Nov/09) resulting in over 45GB of monitoring data. We allocate the buffer pool in 64MB chunks, the storage cache in 128MB chunks, and the disk in 32ms quanta slices, yielding a total of $16 \times 8 \times 8 = 1024$ samples measured for each application. A more accurate solution can be obtained at finer grain increments, e.g., 32MB chunks, but the experiments are estimated to take years in this case.

4.10.2 Benchmarks

We evaluate our techniques using synthetic benchmarks (Workload-A/B/C and UNIFORM) and industry-standard benchmarks (TPC-W, and TPC-C).

**Synthetic Benchmarks**

- **Workload-A/B/C**: We generate three synthetic workloads: small workloads (Workload-A and Workload-C) with 1 outstanding I/O request, and a large workload (Workload-B) with 10 outstanding I/O requests concurrently on the storage server. All are generated using the
ORION\(^1\) (Oracle IO Numbers) tool with different command-line options to generate the variants. Workload-A and Workload-C are cache friendly and Workload-A achieves a cache hit-ratio of 50% with a 1GB storage cache and Workload-C achieves a cache hit-ratio of 100% with a 1GB cache. Workload-B is mostly un-cacheable; it obtains only a 5% hit ratio with a 1GB cache.

▷ UNIFORM: We generate the UNIFORM workload by accessing data in an uniformly random order. The behavior is controlled by two parameters: the size of the data set \((d)\) and the memory working set size \((w)\). We run the workload with \(d=64\text{GB}\) and \(w=1\text{GB}\).

Industry-standard Benchmarks

▷ TPC-W: The TPC-W benchmark from the Transactional Processing Council is a transactional web benchmark designed for evaluating e-commerce systems. Several web interactions are used to simulate the activity of a retail store. The database size is determined by the number of items in the inventory and the size of the customer population. We use 100K items and 2.8 million customers which results in a database of about 4 GB. We use the shopping workload that consists of 20% writes. To fully stress our architecture, we create TPC-W\(^{10}\) by running 10 TPC-W instances in parallel creating a database of 40 GB.

▷ TPC-C/DBT-2: The TPC-C benchmark \cite{80} simulates a wholesale parts supplier that operates using a number of warehouse and sales districts. It simulates a wholesale parts supplier that operates using a number of warehouse and sales districts. Each warehouse has 10 sales districts and each district serves 3000 customers. The workload involves transactions from a number of terminal operators centered around an order entry environment. We scale TPC-C by using 128 warehouses, which gives a database footprint of 32GB. For the evaluation of context-aware prefetching, we use the DBT-2 benchmark, an OLTP workload derived from the TPC-C benchmark \cite{80,110}. We scale DBT-2 by using 256 warehouses, which gives a database footprint of 60GB.

4.10.3 Evaluation Methodology

Our evaluation infrastructure consists of three machines: (1) a storage server running Akash to provide virtual disks, (2) a database server running MySQL, and (3) a load generator for

\(^1\)Oracle ORION – http://www.oracle.com/technology/software/tech/orion/index.html
the benchmarks. We use three workloads: a simple micro-benchmark, called UNIFORM, and two industry-standard benchmarks, TPC-W and TPC-C. In our experiments, the benchmarks share both the database and storage server machines, using the (default) LRU replacement, and containing 1GB of memory each. The buffer pool quotas are allocated in 64MB increments, with a minimum of 64MB. The storage cache quotas are allocated in 128MB increments. Disk quotas are allocated as 32ms (12.5% of the total disk time) disk quanta slices.

We run our Web based applications (TPC-W) on a dynamic content infrastructure consisting of the Apache web server, the PHP application server and the MySQL/InnoDB (version 5.0.24) database engine. We run the Apache Web server and MySQL on Dell PowerEdge SC1450 with dual Intel Xeon processors running at 3.0 Ghz with 2GB of memory. MySQL connects to the raw device hosted by the NBD server. We run the NBD server on a Dell PowerEdge PE1950 with 8 Intel Xeon processors running at 2.8 Ghz with 3GB of memory. To maximize I/O bandwidth, we use RAID 0 on 15 10K RPM 250GB hard disks. In addition, we use the Linux O_DIRECT mode to bypass any OS-level buffer caching and the noop I/O scheduler. We configure Akash to use 16KB block size to match the MySQL/InnoDB block size. Each workload instance uses a different virtual volume: a 32GB virtual disk for TPC-C, a 64GB virtual disk for TPC-W, and a 64GB disk for UNIFORM.

4.10.4 Sampling Methodology

For each hosted application, and given configuration, in order to collect a sample point, we record the average and standard deviation of the data access latency, for the corresponding application in that configuration. For each sample point where we change the cache configuration, we wait for cache warm-up, until the application miss-ratio is stable (which takes approximately 15 minutes on average in our experiments). Once the cache is stable, we monitor and record the application latency several times in order to reduce the noise in measurement. Once measured, sample points for an application can also be stored as an application surface on disk and later retrieved. For the purpose of runtime sampling, i.e., for comparing the predictions made from our performance model to measured samples and to iteratively correct the inaccuracies, the controller iteratively sets the desired resource quotas and measures the application latency during each sampling period.

We use the following rules of thumb in order to speed up the runtime sampling process:

1. Cost-aware Iteration: We sort resources in descending order of re-partitioning cost
i.e., cache repartitioning has higher re-partitioning sampling cost compared to the disk due to the need to wait for cache warm-up in each new configuration. Therefore, we go through all cache partitioning possibilities as the outermost loop of our iterative search; for each cache setting we go through all possible disk bandwidth settings in an inner loop, thus making fewer changes to stateful resources overall.

2. **Order Reversal:** The time to acquire a sample can be further reduced by iterating from larger cache quotas to smaller cache quotas i.e., from 1024MB to 32MB in a 1024MB cache. In this case, the cache warm-up of the largest cache quota will be amortized over the sampling for all cache quotas for the application.

We analyze our sampling methodology in detail in Section 4.12.4.

### 4.11 Results

We evaluate our approach using the **UNIFORM** synthetic benchmark, and the **TPC-C** and **TPC-W** industry standard benchmarks. We first characterize our workloads by preliminary experiments showing their computed MRC at the buffer pool level, then report and compare the average data access latency, measured at the first level cache, for each application, when using different resource partitioning schemes.

#### 4.11.1 Miss-Ratio Curves

Figure 4.6 shows the **miss-ratio curves** at the first level cache (buffer pool) for all applications. We can see that **TPC-W** and **TPC-C** are more cacheable than **UNIFORM**. **UNIFORM** has comparatively higher miss-ratios, and it benefits greatly from larger cache allocations. On the other hand, **TPC-W** and **TPC-C** are less affected by cache allocations past 128MB.

#### 4.11.2 Running Two Applications

We run either identical workload instances, or different workload instances, concurrently, on our infrastructure, and compare the performance of our partitioning algorithms. We summarize the relative performance of each algorithm compared to the **IDEAL** scheme in Table 4.2. The details of each experiment is presented in Figures 4.7 to 4.12 – where each figure shows the latency and configuration chosen for each application by the various resource partitioning schemes.
We notice the following overall trends in our results. Our GLOBAL+ partitioner matches the performance of the IDEAL* scheme (in 5 out of 6 experiments), at a fraction of the cost. The performance of the GLOBAL partitioner, based only on the computational model, is relatively close to the ideal performance as well; GLOBAL registers improvements with experimental sampling only for workload combinations that include TPC-C, an application with a substantial fraction of writes. We find that adding 256 runtime performance samples to TPC-C is sufficient to obtain near-optimal performance. Moreover, with one exception, our GLOBAL partitioner is both faster and generates better partitioning settings than the combination of single resource controllers i.e., the MRC+DISK partitioner.

The single resource partitioning schemes, i.e., MRC and DISK, are limited in their ability to control performance. For example, DISK is ineffective for cache-bound workloads (see Figures 4.7, 4.10, 4.11). A more subtle point is that in some cases, the poor choices made by the MRC scheme can be corrected by providing more disk bandwidth to disadvantaged applications in the MRC+DISK scheme. We discuss our performance results in detail next and we examine the accuracy of our model and its refinements in Section 4.12.3.
### Running Two Identical Applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Performance Relative to IDEAL*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GLOBAL</td>
</tr>
<tr>
<td>UNIFORM</td>
<td>1.00</td>
</tr>
<tr>
<td>TPC-W</td>
<td>1.05</td>
</tr>
<tr>
<td>TPC-C</td>
<td>1.06</td>
</tr>
</tbody>
</table>

### Running Two Different Applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Performance Relative to IDEAL*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GLOBAL</td>
</tr>
<tr>
<td>UNIFORM</td>
<td>1.00</td>
</tr>
<tr>
<td>TPC-W</td>
<td>1.16</td>
</tr>
<tr>
<td>TPC-C</td>
<td>1.80</td>
</tr>
</tbody>
</table>

Table 4.2: We summarize the performance of different resource allocation algorithms compared to the performance of the IDEAL* scheme. The best result is highlighted in **bold**.

**Running Two Identical Applications**

First, we look at cases where we run two instances of the same application, so the applications have the same miss-ratio curves leading the MRC scheme to partition both the database buffer pool and the storage cache equally between the two applications. With such settings, the second level cache is wasted due to cache inclusiveness resulting in poor performance. The same reason applies to the poor performance of the DISK scheme – that always partitions the caches equally between the applications. In the following, we explain the details of each experiment.

Figure 4.7 presents our results for the UNIFORM/UNIFORM configuration. Our GLOBAL scheme finds a resource partitioning setting of 64MB/960MB and 960MB/64MB between the two instances of UNIFORM, at the buffer pool and storage caches respectively. This setting provides a much better cache usage than equal partitioning of the two caches. Overall, GLOBAL provides the same partitioning settings as IDEAL* and obtains a factor 5.98 speedup over MRC+DISK.

We also experiment with running two identical instances of TPC-W and TPC-C. The results are shown in Figure 4.8 and Figure 4.9. Similar to the first case of running two UNIFORM workloads, the applications have the same miss-ratio curve leading the MRC scheme to partition the caches roughly equally between the two applications in the first-level cache. For example, the first-level cache is partitioned into 448/576 for TPC-W and 384/640 for TPC-C.
Due to this poor decision by the MRC allocator, the performance of the MRC scheme is a factor of 1.40 and 1.42 worse than the IDEAL scheme for TPC-W and TPC-C respectively. The DISK scheme performs poorly as it wastes cache resources that are critical to improving performance of these applications. This results of the DISK scheme performing a factor of 1.58 and 1.11 worse than IDEAL for TPC-W and TPC-C respectively.

While each performing poorly on its own, the combination of MRC and DISK yields better performance, as allocating the bandwidth after allocating memory allows for the correction of the faults in the MRC scheme. For example, when running the TPC-W workloads, the combination scheme MRC+DISK obtains a performance that is a factor of 1.29 worse than IDEAL (when MRC is a factor of 1.40 worse and DISK is 1.58 worse), providing improvements of 10% and 22% over single-resource allocators. However, the improvement is not possible in all cases as seen in the experiment running TPC-C where the MRC+DISK scheme fails to improve on MRC scheme.

The GLOBAL scheme is able to model TPC-W well, achieving near-optimal performance without any runtime performance samples; we added 64 samples for TPC-W in GLOBAL+ but do not see any further improvements. Our models are not as accurate for the write-intensive TPC-C application, leading us to add 256 runtime performance samples to refine the initial model. With the refined model, our GLOBAL+ scheme, by properly accounting for the interdependency between resources, is able to achieve much better performance; it is approximately 5% worse from the performance obtained by the IDEAL scheme and a factor 1.23 better than the MRC+DISK scheme.

Running Two Different Applications

Figures 4.10-4.12 present our results for different concurrent workloads. These results show that the allocations chosen by the GLOBAL partitioner are non-trivial, and good performance is obtained only when the settings of all resources are considered. Single resource partitioning, as in the MRC and DISK partitioners, tend to overlook the fact that application performance is affected not only by each resource but that the behavior of a resource is affected by other upstream resources. Finally, poor choices made by the MRC scheme can be corrected by using the MRC+DISK scheme. However, the benefits of the correction are limited. We next discuss each configuration in detail.

First, we examine the TPC-W/UNIFORM configuration, shown in Figure 4.10. The UNIFORM workload has both larger cache and disk requirements than TPC-W. Since the miss-ratio curve of UNIFORM is steeper than that of TPC-W, once the first 64MB is allocated to TPC-W, the MRC
Running Two Instances of UNIFORM. We run two instances of the UNIFORM workload. With identical miss-ratio curves, the MRC scheme divides the caches equally reducing it benefit. The storage cache does not provide any additional benefit due to cache inclusiveness. The GLOBAL scheme finds the near-optimal solution while the MRC+DISK is a factor of 5.98 worse than the IDEAL*. 

```
<table>
<thead>
<tr>
<th>Scheme</th>
<th>Buffer Pool</th>
<th>Storage Cache</th>
<th>Disk Quanta</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UNIFORM</td>
<td>UNIFORM</td>
<td>UNIFORM</td>
</tr>
<tr>
<td>GLOBAL</td>
<td>64 960</td>
<td>960 64</td>
<td>50 50</td>
</tr>
<tr>
<td>GLOBAL+</td>
<td>64 960</td>
<td>960 64</td>
<td>50 50</td>
</tr>
<tr>
<td>MRC</td>
<td>512 512</td>
<td>512 512</td>
<td>50 50</td>
</tr>
<tr>
<td>DISK</td>
<td>512 512</td>
<td>512 512</td>
<td>50 50</td>
</tr>
<tr>
<td>MRC+DISK</td>
<td>512 512</td>
<td>512 512</td>
<td>50 50</td>
</tr>
<tr>
<td>IDEAL*</td>
<td>64 960</td>
<td>896 128</td>
<td>50 50</td>
</tr>
</tbody>
</table>
```
Running two instances of TPC-W. We run two instances of the TPC-W workload. The MRC scheme divides the buffer pool roughly equally thus reducing its benefit. The MRC+DISK corrects the faults of the MRC leading to better performance. However, both schemes perform poorly resulting in performance that is factors of 1.40 and 1.29 worse than the IDEAL*.

On the other hand, the GLOBAL scheme is closer and is only a factor of 1.05 worse than the IDEAL*. 

Figure 4.8: Running two instances of TPC-W.
Figure 4.9: **Running two instances of TPC-C.** We run two instances of the TPC-C workload. The TPC-C workload is write-intensive causing our approximate performance models to have more errors. This causes the GLOBAL to achieve less than ideal performance (a factor of 1.06 worse than IDEAL*). The performance can be improved by iterative refinement using the GLOBAL+ scheme which improves the performance from 1.06 to 1.04. The MRC/DISK/MRC+DISK perform worse by not considering the interdependence between resources.
partitioner allocates the rest of the buffer pool (960MB) to UNIFORM. The DISK partitioner selects a 75/25 disk bandwidth allocation in favor of UNIFORM. But, dividing the caches 50/50 results in poor performance for this partitioner. The MRC+DISK scheme is not able to correct the poor choice of the cache partitioning resulting in performance equal to the MRC scheme. The computed models are accurate and do not need any runtime correction. The GLOBAL scheme performs a factor of 1.13 better than MRC+DISK, by obtaining a better cache configuration overall. By properly allocating the caches, the GLOBAL scheme performs a factor of 1.13 better than MRC, and a factor of 3.80 better than DISK.

Next, we look at the TPC-C/UNIFORM configuration, shown in Figure 4.11. The results are similar to the TPC-W/UNIFORM configuration, with one exception. The model for our GLOBAL partitioner mispredicts the cache behavior at the storage cache for TPC-C. The assumption about block redundancy between the buffer pool and storage cache does not hold for TPC-C, an application with a substantial fraction of writes. Due to this, allocating more storage cache to TPC-C as chosen by MRC/DISK/MRC+DISK is beneficial resulting in increased hit rates in this cache. We try to deduce this behavior by adding up to 256 runtime performance samples but the performance does not improve leading the GLOBAL+ scheme to perform equal to the GLOBAL scheme. The DISK partitioner under-performs for the same reason as before i.e., because allocating the cache resources 50/50 penalizes UNIFORM. The MRC+DISK corrects the disk fraction resulting in performance equal to IDEAL+. The GLOBAL+ performs a factor of 1.05, and 3.59 better than MRC, and DISK, respectively, and slightly less than the MRC+DISK scheme. This is the only experiment where the GLOBAL+ scheme did not perform the best.

Finally, we study the TPC-W/TPC-C configuration, shown in Figure 4.12. As the miss ratio curve for TPC-C is slightly steeper than TPC-W, the MRC partitioner allocates a slightly larger fraction of the buffer pool (576MB) to TPC-C. Moreover, the miss ratio curves of the two applications are similar to each other at the storage cache level. Therefore, the same greedy MRC cache algorithm allocates a larger fraction of the storage cache (768MB) to TPC-C as well. This results in over-allocation of total cache space to TPC-C, penalizing TPC-W, when compared to the cache configuration and performance achieved by IDEAL+. Allocating a larger disk fraction to TPC-W in MRC+DISK compensates for the poor cache partitioning of MRC alone.

The GLOBAL scheme chooses a solution following our belief that, due to cache inclusiveness, caches must allocated asymmetrically for good performance – that is, if we allocate most of the buffer pool to one application then we should allocate less at the storage cache.
Figure 4.10: Running TPC-W and UNIFORM. We measure the performance of TPC-W and UNIFORM running concurrently. The DISK scheme chooses the poor allocation for the UNIFORM workload leading to a large performance degradation. The MRC makes the right choice at the first-level but a poor choice at the storage cache by not accounting for cache inclusiveness. The GLOBAL performs a factor of 1.13 better than MRC and a factor of 3.80 better than the DISK scheme.
Figure 4.11: Running **UNIFORM** and **TPC-C**. We measure the performance of **UNIFORM** and **TPC-C** running concurrently. As before, the **DISK** scheme chooses the poor allocation for the **UNIFORM** workload leading to a large performance degradation. The **MRC** partitions the caches correctly at both levels and with the disk correction, the **MRC+DISK** scheme achieves performance equal to the **IDEAL** scheme. The **GLOBAL** performs a factor of 1.05 better than **MRC** and a factor of 3.59 better than the **DISK** scheme. In this case, the **MRC+DISK** performs the best matching the performance of the **IDEAL** scheme.
Running TPC-W and TPC-C. We measure the performance of TPC-W and TPC-C running concurrently. As the miss-ratio curves of the two workloads are similar, the MRC scheme partitions the caches roughly equally at both levels leading to poor performance. The DISK scheme makes a poor choice for allocating the caches. The GLOBAL+ performs a factor of 2.15 better than MRC and a factor of 1.91 better than the DISK scheme.
Figure 4.13: Running Four Application Instances. We run four instances of TPC-W and TPC-C. In both cases, the MRC/DISK/MRC+DISK make the identical and poor choice to partitioning the resources equally in both levels of the cache resulting in high latencies for all applications. Our GLOBAL+ scheme allocates the caches more effectively leading to improved performance.

With this belief, GLOBAL allocates most of the buffer pool to TPC-W and most of the storage cache to TPC-C. However, GLOBAL+ reverses the settings chosen by GLOBAL because TPC-C is able to utilize the storage cache better due to its more write-intensive nature. The GLOBAL scheme using our approximate model cannot model this beneficial effect of writes leading it to allocate a larger fraction of the buffer pool to TPC-W. By adding 256 runtime performance samples for TPC-C we refine the model and arrive at a better resource partitioning setting. As a result, GLOBAL+ improves by a factor of 2.15 better than MRC, a factor of 1.91 better than DISK, and a factor of 2.03 better than MRC+DISK.

### 4.11.3 Running Four Applications

To evaluate the scalability of our approach, we also experiment with running four applications concurrently on our infrastructure. We run four identical instances for simplicity. Figure 4.13 presents our results when running four instances of the TPC-W and TPC-C workloads. Similar to the case of running two identical applications, the miss ratio curves of the four applications are identical leading the MRC/MRC+DISK/DISK schemes choose to partition the cache levels equally at both the client and storage caches. With this setting, due to cache inclusiveness, the second level cache, i.e., the storage cache, provides little benefit, resulting in poor performance for these partitioners. The above schemes are a factor of 1.56 worse than the IDEAL* for the TPC-W workload and a factor of 3.38 worse for TPC-C.
Figure 4.14: Running Applications to Maximize Revenue. We run two instances of the TPC-W workload where one of the instances is designated to be high-priority and the other is marked as a best-effort application. By being SLO aware, the GLOBAL scheme generates a profit while other SLO agnostic approaches lead to penalties.

For TPC-W, as shown in Figure 4.13a, our GLOBAL scheme, finds resource partitioning settings of \{(64MB/384MB), (64/384), (448/128), (448,128)\} for the four instances of TPC-W, at the buffer pool and storage caches respectively. This leads to a near-optimal performance that is only a factor of 1.08 worse than the IDEAL\textsuperscript{*} scheme and a factor of 1.44 better than the MRC+DISK scheme. Adding more performance samples does not improve the allocation leading the GLOBAL\textsuperscript{+} scheme to perform equally as well as the GLOBAL scheme. For the TPC-C workload, the GLOBAL scheme finds a solution that is a factor of 1.57 worse than the IDEAL\textsuperscript{*}. This is due to the fact that our approximate performance models cannot model the effect of writes well. By correcting this error using runtime samples, the GLOBAL\textsuperscript{+} improves by achieving performance that is only a factor of 1.19 worse than the IDEAL\textsuperscript{*} scheme. This results in GLOBAL\textsuperscript{+} to perform a factor of 2.84 better than the MRC/DISK/MRC+DISK schemes.

4.11.4 Resource Allocation with Utility Functions

In our evaluation, we use the combined application latencies (by simple summation) as the global optimization goal. We choose this simple metric for fairness of comparison with the miss-ratio curve algorithm [115], which optimizes the aggregate miss-ratio, hence the aggregate latency, while being agnostic to service level objectives (SLOs) in general.

In this section, we illustrate how general utility functions can be handled using our approach and how local optimization goals, i.e., minimizing miss-ratio at each cache, can
run counter to the overall performance goal of maximizing utility. We show this by repeating the experiment with two identical instances of TPC-W: TPC-W (T-A) marked as a high-priority application and TPC-W (T-B) marked as a best-effort application. We express the SLO in terms of average data access latency. A data access latency SLO of less than 500\(\mu\)s provides an average query response time below 500ms for our benchmarks, which closely approximates values used as QoS for the two e-commerce applications in previous studies [96]. Thus, in our utility function, we set \(D = 500\mu\)s, \(D' = 4000\mu\)s (equivalent to the underlying disk access time), \(U_{min} = -100\) and \(U_{max} = 100\).

Figure 4.14a shows the data access latency of both TPC-W instances with different resource partitioning schemes. The MRC scheme performs poorly as it does not have a notion of SLOs causing T-A to violate its deadline; the latency of T-A has 0.82ms latency and the T-B has 1.52ms latency. A similar problem occurs with the DISK scheme resulting in a latency of 1.32ms for both T-A and T-B. In all these schemes, by being unaware of the application-level SLOs, the performance is poor resulting in a penalty for the provider; as shown in Figure 4.14b, the penalty for MRC is -20, -157 for DISK, and -92 for MRC+DISK.

On the other hand, by being aware of the SLOs and application priorities, our GLOBAL scheme allocates more resources to T-A that allows it to meet the deadline of 0.5ms and it allocates the remaining resources to T-B. To illustrate how the utility function changes the resource allocation, we present the latencies for the GLOBAL scheme without the utility function as well. As Figure 4.14b shows, by designating T-A as a high-priority causes the GLOBAL scheme to allocate more resources to T-A. This leads to the drop in the latency from 0.70ms to 0.49ms and a corresponding latency increase of T-B from 1.06ms to 1.33ms. By considering the utility functions and by searching for configurations to maximize revenue (utility), the GLOBAL scheme generates a profit of 76 as opposed to generating penalties as in the SLO agnostic schemes.

### 4.12 Detailed Analysis

In this section, we evaluate the accuracy of our cache and disk approximations in our performance model. In addition, we present results for online refinement of our model through experimental sampling.
Figure 4.15: **Two-level Cache Approximation – Curves.** Errors for cache configurations with TPC-W and TPC-C. Figures 4.15a and 4.15b shows the measured and predicted miss ratio curves for buffer pool sizes (64MB, 256MB, 512MB).

Figure 4.16: **Two-level Cache Approximation – HeatMap.** We show error heatmaps where light/dark colors represent low/high error, respectively. The magnitude of the error is shown in the legend on the right.

### 4.12.1 Accuracy of the Two-Level Cache Model

We evaluate the accuracy of the two-level cache miss-ratio prediction. Figure 4.15 presents our results for TPC-W and TPC-C. We first provide a detailed analysis for TPC-W, for three buffer pool size (64MB, 256MB, 512MB) and a range of storage cache sizes, where we plot two cache miss ratio curves: experimentally measured (solid lines) and predicted by model (dashed lines). As we can see, the predicted and measured miss ratio curves are close
Figure 4.17: **Accuracy of Quanta Scheduler Approximation.** We plot the predicted and measured disk latency by varying the disk scheduler quota, in different cache configurations, from 128MB cache to 960MB cache.

...together, hence, our cache approximation is accurate in calculating the miss ratio at the storage cache. The areas of inaccuracy, where the relative error is greater than 2%, occur when the storage cache is equal to the buffer pool size i.e., 512M. The replacement policy is affected by concurrency control i.e., through the fix/unfix of buffer blocks and some other thread optimizations to mitigate cache pollution for table scans, in this case.

We further present the error of our model as a more general heat-map, where low errors (0-20%) are shown in light colors, whereas higher errors are shown in darker colors, for a wide range of cache configurations, for both our benchmarks. For both benchmarks, the area of any significant inaccuracy is where the two cache sizes are equal, especially for large cache sizes. However, these very configurations are unlikely to be used as an allocation solution, because they correspond to a high level of redundancy for uncoordinated two-level LRU caches. Moreover, for high cache sizes, the miss ratio of most applications is low, hence the error is less relevant. The errors are higher for TPC-C due to its large fraction of writes, hence unpredictable hits in the storage cache for dirty blocks previously evicted from the buffer pool. For both benchmarks, the error falls below 2% when the storage cache is at least a factor of 2 larger than the buffer pool size.
Figure 4.18: Iterative Refinement with Runtime Sampling. We refine our model accuracy at runtime with experimental sampling. A minimum of samples is required to correct the models for TPC-W and UNIFORM. TPC-C requires more samples due to its write-intensive nature.

4.12.2 Accuracy of the Quanta-based Scheduler Approximation

We evaluate the accuracy of our disk latency approximation, when using a quanta-based scheduler. We plot both the predicted and the measured disk latency, for each application, by varying the storage bandwidth quanta. Figure 4.17a and Figure 4.17b present our results for TPC-C and TPC-W, respectively. In each graph, we plot and compare two lines: measured (solid lines) and predicted (dashed lines), for different cache sizes (given mostly at the buffer pool).

Overall, the predicted disk latency deviates significantly from the measured latency only for small quanta values. Moreover, slightly higher errors can be observed for higher cache sizes. In both of these cases, the explanation is that there is a higher variability in the average disk latency over time when (i) the underlying disk bandwidth isolation is less effective due to frequent switching between workloads and (ii) disk scheduling optimizations are less effective and reliable due to fewer requests in the scheduler queue.

4.12.3 Model Refinement with Runtime Sampling

As shown, our model is inaccurate in very localized areas of the total search space, where inaccuracies may not matter, or can be improved by experimental sampling. Figure 4.18 shows the accuracy improvement that is obtained through online performance sampling. We define the prediction error as the relative difference between the predicted latency obtained from the model and the measured latency obtained by sampling the search space.
We calculate the error by separating the performance samples into two parts: a training set (consisting of 768 randomly chosen configurations) and a testing set (consisting of the remaining 256 configurations). We train the model using different amounts of performance samples to correct the initial approximate model; a sampling amount of zero indicates no runtime correction. We then use this trained model to predict the latencies for the rest of the 256 configurations and compute the error as the difference between the predicted values and the measured values. Figure 4.18 presents our results. In the \( x \)-axis we show the number of samples added to our performance model experimentally, and on the \( y \)-axis we show the relative error between the predicted and measured values. As we have seen before, our approximate performance models of TPC-W and UNIFORM are accurate in most of the configurations; we need a few runtime samples to correct the model in problematic configurations when the cache sizes are roughly equal and when the disk quanta is small. This is seen in the figure where adding more than a minimum amount of samples does not improve the quality of the performance model. On the other hand, the TPC-C workload is harder to model due to its write-intensive nature. The error for TPC-C steadily reduces with more runtime samples. We however note, as we did in previous sections, near-optimal solutions can be obtained with less than fully-accurate models.

4.12.4 Cost of Obtaining Experimental Samples

We have shown that by adding performance samples obtained from the running system we can correct the errors in the approximate performance model. We obtain performance samples iteratively, as described in Algorithm 1, obtaining \( k \) samples in each iteration. In addition, we implement several optimizations such as a cost-aware iteration – where we sample configurations in descending order of re-partitioning cost and order reversal – where we sample cache sizes from larger cache sizes to smaller cache sizes. These optimizations enable us to obtain performance samples in a reasonable amount of time. However, the optimizations only help if we fetch a sufficient number of samples, defined by \( k \), to amortize the sampling cost – that is, there is an inherent tradeoff (controlled by \( k \)) where obtaining a large set of samples reduces the cost per sample but increases the cost per iteration.

Figure 4.19 presents our results. The costs are for the sampling algorithm with our optimizations enabled but the applicability of the optimizations is controlled by the sampling size, \( k \), defined earlier. In the left figure, we show the time needed to obtain one sample (\( y \)-axis) for different sampling amounts (varying \( k \) in the \( x \)-axis). The figure shows that obtaining performance samples one at a time is costly, taking roughly 5 minutes per sample,
Figure 4.19: **Cost of Runtime Sampling.** We plot the cost (time) to obtain samples from the running system. We iteratively obtain samples for each application correct errors in the approximate performance model at runtime. We use several optimizations to lower this cost where the cost to obtain a large set of samples is faster but each iteration step can take a long time.

and by obtaining more samples, we can amortize the cost. With increasing $k$, the cost per sample asymptotically approaches 2 minutes. The figure on the right shows the duration of each sampling iteration where obtaining a small number of samples per iteration takes shorter time than the time to obtain a large set of samples.

We use these two results to determine the value of $k$ in our experiments. We set the value of $k$ to 16 for the TPC-W and UNIFORM workloads and we set the value of $k$ to 32 for the TPC-C. Generally, choosing values for $k$ between 16 to 64 provides a good tradeoff – it keeps the per-iteration cost under 120 minutes (2 hours) and reduces the per-sample cost by half. For the read-intensive workloads, i.e., TPC-W and UNIFORM, our models are accurate thus we can use the 16 runtime samples to verify the accuracy of our analytical performance model. On the other hand, our model is less accurate for TPC-C, i.e., requires more runtime samples, leading us to obtain more samples per iteration to amortize the cost. With these optimizations, the time needed to correct the TPC-C model is roughly 640 minutes (256 samples). We need roughly 40 minutes (16 samples) to verify the approximate models for the TPC-W and UNIFORM workloads. Further optimizations can be done such as varying $k$ at runtime based on the model’s accuracy. We leave this optimization for future work.
4.13 Summary

Resource allocation to applications on the fly is increasingly desirable in shared data centers with server consolidation. While many techniques for enforcing a known allocation exist, dynamically finding the appropriate per-resource application quotas has received less attention. The challenge is the exponential growth of the search space for the optimal solution with the number of applications and resources. Hence, exhaustively evaluating application performance for all possible configurations experimentally is infeasible.

Our contribution is an effective multi-resource allocation technique based on a unified resource-to-performance model incorporating (i) pre-existing generic knowledge about the system and inter-dependencies between system resources e.g., due to cache replacement policies and (ii) application access tracking and baseline system metrics captured on-line.

We implement our global resource allocator within a virtual storage prototype with a two-tier cache hierarchy. We leverage the Network Block Device (NBD) to integrate our prototype within existing commodity software and hardware environments i.e., the MySQL database engine and a commodity servers and storage hardware, with no changes to interfaces between components and minimal DBMS instrumentation. We show through experiments using several standard e-commerce benchmarks and synthetic workloads that our performance model is sufficiently accurate in order to converge towards a near-optimal global partitioning solution with minimal runtime sampling. Our results show that multi-resource partitioning allows for performance improvements of up to a factor of 6 for synthetic benchmarks, and a factor of 4 for industry-standard benchmarks compared to state-of-the-art single-resource controllers, and their ad-hoc combination. At the same time, the results show that our techniques achieve performance that is within 20% of the performance achieved by an exhaustive approach, but in a fraction of the time.
Chapter 5

Validating System Performance

In this chapter, we study high-level paradigms and advanced tools for helping the administrator validate system performance. We introduce *SelfTalk*, a novel declarative language and *Dena*, a runtime support system to help system administrators diagnose and validate the performance of a multi-tier system quickly and accurately.

5.1 Introduction

As modern systems become exceedingly large and complex, and their applications increasingly sophisticated, system developers and administrators find themselves in the difficult situation of not being able to validate the performance and behavior of the system easily anymore. Thus, expanding the use of automated tools to monitor large-scale multi-tier systems has become a life-saving goal for the computer industry. Indeed, many commercial tools for monitoring and control of large-scale systems exist; tools such as HP’s Openview and IBM’s Tivoli products collect and aggregate information from a variety of sources and present this information graphically to administrators. However, the complexity of deployed systems exceeds the ability of humans to validate the performance and behavior and to diagnose and respond to problems rapidly and correctly [53].

These existing approaches for automated performance diagnosis and validation either (i) compare the performance to full *analytical* models describing the system structure and behavior [46, 89, 101], or (ii) compare the current performance to *probabilistic* models derived from instrumenting the system [21, 38, 47]. Both approaches can be applied to a wide variety of systems, and can adapt rapidly to system changes (albeit with some re-training time). However, the primary obstacle to widespread adoption of these approaches is the
lack of context-awareness – that is, these approaches do not take advantage of the generic environmental conditions, such as, configuration parameters, information about the applications, the structure of the system, or the administrator’s experience and beliefs. Without this knowledge, these approaches may trigger unacceptable levels of false alarms for benign changes, such as, a workload mix change or an environmental setting change. Indeed, the interdependency between components in a multi-tier system is evident; in our previous work (described in Chapter 4), we have shown the effect of cache inclusiveness in multi-tier caches and the effect of the cache partitioning and bandwidth allocation on application performance.

In this dissertation, we take the first step to incorporate context-awareness to improve the methods of monitoring, diagnosing, and validating the performance of multi-tier systems. Towards establishing the necessary dialogue between the system administrator and the self-managed system, as a cornerstone of their symbiotic relationship, we make the following contributions. We introduce (i) a new high-level declarative language, called SelfTalk and (ii) new runtime support system, called Dena, capable of learning, self-monitoring its metrics, and evolving dynamic models of metric correlations, as well as interacting with its administrator in SelfTalk.

With SelfTalk, the system administrator can, for the first time, easily express her beliefs and expectations about what constitutes normal system behavior, indicate the parameters and contexts affecting that behavior, ask the system to validate those beliefs within given periods of time, query the system status at any point in time, and obtain meaningful responses. Our language is powerful and can encode known generic laws that govern system behavior, such as Little’s law [46] that correlates throughput and latency values, or the monotonically decreasing property of the miss-ratio curve (MRC) [115] in any system cache, known relationships between any metric classes, such as, the expectation of an exponential correlation between latency and the resource quotas allocated to an application, as well as more specific administrator insights, experience with the system, or a given application. For example, SelfTalk allows the human administrator to express her beliefs in close to the following high-level format: “I expect that the average query latency measured at the database system is greater than the average data access latency measured at its backend storage server”. The beliefs do not need to be always correct, and should be viewed more as expressed hypotheses to the system, rather than rigid assertions.

Dena parses the administrator’s hypothesis, constructs a concrete mathematical expression to describe the relationship between metric classes, and evaluates compliance by fitting
the accumulated monitoring data from the system within any given context to the mathematical expression. The system thus specializes the high level hypotheses into concrete internal expectations that match each hypothesis within a particular context, and enters the hypothesis, the matching expectations, contexts, and the confidence score into a knowledge base. SelfTalk/Dena thus provides the basis for evolving system self-expression as the human-like ability to agree(validate), or disagree with the system administrator on facts and beliefs about the system in relation to given environments/contexts. As in humans, development of self-expression is incremental, as new guidance is received from the administrator, and new situations occur, where the guidance received is applicable and valid, or not. The status of the multi-tier system relative to any and all hypotheses can be queried at any point in time; Dena outputs to the administrator its degree of confidence that the system conforms to any given hypothesis, by checking whether its monitored data fits the hypothesis well. Thus, in situations of misbehavior, we argue that interacting with Dena allows the administrator to detect anomalies and thus diagnose the root cause quickly.

We perform an evaluation of our approach by posing several hypotheses to understand normal behavior of the overall system, to understand the behavior of individual components, and to diagnose misbehavior in a multi-tier dynamic content server, consisting of an Apache/PHP web server, a MySQL database using virtual volumes, hosted on a virtual storage system called Akash [98], running the industry-standard benchmarks, TPC-W and TPC-C. We find that Dena can quickly validate system performance to users’ hypotheses and can help in diagnosing faults, or other system misbehavior. In addition, we validate different components, such as the MySQL cache, the storage system I/O scheduler, and the behavior of multi-level caches using Dena. Dena provides these results quickly, by returning replies under 2 seconds on approximately 50GB of monitoring data, collected over a period of 11 months.

The rest of the chapter is organized as follows. Section 5.2 presents an overview of SelfTalk/Dena. We describe the SelfTalk language in Section 5.3. Section 5.4 describes the different steps taken by Dena to filter the monitoring data, to apply statistical regression, and to compute the confidence score. Section 5.5 provides examples of hypotheses issued to understand a multi-tier storage system. Section 5.6 provides the details of our prototype implementation. Section 5.7 presents our evaluation infrastructure and Section 5.8 presents results of our analysis. Section 5.9 concludes the chapter.
5.2 Architectural Overview

We introduce a novel language and runtime support for understanding and validating the behavior of a multi-tier dynamic content server system. Specifically, we design a high-level declarative language, called *SelfTalk*, that allows the system administrator\(^1\) to express generic hypotheses about normal system behavior, including known system laws, and relationships between metric classes. The system administrator submits *hypotheses* to a runtime system, called *Dena*, that instantiates and validates them using metrics collected from various components of the multi-tier system. In the following, we describe the *SelfTalk* language, the design of *Dena*, our tool, and how the administrator and the system interact to check the validity of her beliefs and the compliance of the system.

5.2.1 The SelfTalk Language

A *hypothesis* describes a relationship on a set of metrics, and the associated descriptions of generic environmental conditions, defining a validity *context* for that relationship. The context can be a set of configurations, or workload properties that could potentially affect the given relationship. If the relationship is believed to be an *invariant*, then its corresponding context is empty. *SelfTalk* is an SQL-like language we build to express these hypotheses. In the following, we provide some examples of hypotheses, written in *SelfTalk*, to illustrate these concepts and to highlight the simplicity of the language, and its ease of use.

Consider a scenario where the administrator wants to verify the behavior of a cache in the multi-tier system; this cache may be located within the storage system, or in a database system, or it can be a dedicated caching service. Regardless of its type and location, the administrator can have certain generic beliefs about the cache, such as, the belief that the number of cache misses (*num_cache_miss*) must be less than or equal to the number of cache accesses (*num_cache_gets*), as shown in Listing 5.1. This generic hypothesis, issued by the administrator, expresses her belief about the behavior of a cache, and illustrates an important point about its high level behavior. The administrator does not have to know the details or the inner workings of the cache. She does not have to have access to its source code, or its replacement policy. In fact, this observation is an *invariant* of the cache i.e., it must hold true for all configurations and workloads. Thus, the administrator can submit the hypothesis without a context, and *Dena* will check if this relationship is indeed valid for all configurations.

\(^1\)We use the terms administrator and user interchangeably.
Listing 5.1: Invariant Hypothesis. It expresses the administrator’s belief that the number of cache misses must be less than or equal to the number of cache accesses. This is an invariant of the cache that must hold true for all workloads and configurations.

Listing 5.2: Hypothesis with a Context. It expresses the belief that the throughput-like metrics must be linearly related for cache sizes less than 512MB. The definition of the cache size defines the context for the hypothesis.

Some hypotheses are valid only for particular configurations. For example, in a database system, as the rate of queries processed increases, so does the rate of operations within the operating system, i.e., more I/Os per second but only for configurations where not all data is cached. In this case, the database administrator of this system can hypothesize: “I expect that the throughputs of the different components are linearly correlated”. In Listing 5.2, we show how the above hypothesis is specified in Dena. It states that the throughput metrics, i.e., those with units \(1/s\), are expected to be linearly correlated in configurations where the \(cache_size\) is less than or equal to 512MB.

The above two examples illustrate the simplicity of the SelfTalk language. With SelfTalk, the administrator can express her beliefs about the behavior and performance of a component, the interactions between components, and even the overall behavior of the system. We further reduce the effort of using SelfTalk, by providing simple relations (such as the \(\text{LINEAR}\) and \(\text{LESS-EQ}\) shown above) and pre-built hypotheses for common three-tier components, e.g., Apache and MySQL. However, a more advanced user may define new metrics to monitor, create new relationships to test, and explore new facets of large multi-tier systems. We explain the different features of the SelfTalk language in detail in Section 5.3.
5.2.2 The Dena Runtime System

In the following, we provide the steps taken by Dena when the administrator submits a hypothesis to the system.

1. The system automatically instantiates the hypothesis and generates a (much larger) set of expectations, by enumerating all possible metrics within the metric classes and configurations that match the hypothesis.

2. The system then validates each expectation with monitoring data, computes a confidence score per expectation and stores the expectations in a database; after this step, the system is ready for subsequent analysis.

3. The administrator may now submit a wide variety of queries to Dena, including querying the validity of expectations over components in a sub-part of the system, confidence intervals, number of expectations generated, standard deviations, etc.

In more detail, Dena takes a given hypothesis and creates a list of expectations by iteratively applying the hypothesis for each metric matching the qualifiers, $\vec{Q}$. For a set of metrics, $\mathcal{M}$, Dena extracts a subset of metrics $m_i \in \mathcal{M}$ such that $m_i$ matches all conditions specified in qualifier set $\vec{Q}$. For example, for the query described in Listing 5.2, Dena applies the hypothesis to all throughput metrics creating a list of expectations. In this list, one expectation would be $\text{EXPECT HYP-LINEAR} (x, y) (\text{'name=queries_per_sec'}, \text{'name=io_per_sec'}).$

Second, Dena selects a function that matches the relationship described in the hypothesis. If the relationship is $\text{LINEAR} (\text{'name=queries_per_sec'}, \text{'name=io_per_sec'})$, then we match it with a function $y_{\alpha,\beta}(x_t) = \alpha x_t + \beta$, and instantiate the expectation. Third, Dena takes each expectation, and fits the function to the monitoring data. The curve is fit using an optimization algorithm, i.e., gradient descent, by varying the free parameters in the function. In particular, for the linear correlation between the database and storage system throughput, the curve fitting algorithm searches for values of $\alpha$ and $\beta$ that minimize the squared error from the measured values. The curve fitting algorithm outputs a confidence score, $\gamma$, between $0 \leq \gamma \leq 1$ representing a goodness of fit, where $\gamma = 1$ is a good fit, and $\gamma = 0$ is a poor fit.

Dena provides the aggregate confidence score for the hypothesis, and it allows the user to zoom-in to get per-context confidence scores as well. In the following sections, we provide a detailed description of the SelfTalk language and the details of the Dena runtime system.
5.3 Statements in the SelfTalk Language

In this section, we describe how a hypothesis can be declared in the SelfTalk language and how the generated expectations can be subsequently analyzed using our query language. The SelfTalk language has two types of statements: a hypothesis and a query.

The hypothesis states the administrator’s belief about the behavior of the system; it is identified by a unique name, a relation that describes a relationship between metrics, and a context that indicates the environment factors e.g., configuration parameters, affecting the validity of the hypothesis. Dena processes the submitted hypothesis and provides results on whether or not the administrator’s beliefs match the system’s behavior.

SelfTalk also allows the administrator to query and check the validity of the expectations. Specifically, the administrator can query about the confidence of the expectations (resulting from the expansion of a hypothesis), evaluate the fit under various contexts, and for different sub-components. In addition, the administrator can obtain averages, rank the expectations, and statistically analyze the results computed by Dena. In the following, we describe how a hypothesis can be expressed in SelfTalk. We focus on: i) how to specify the metrics, ii) how to define the relation, and iii) how to specify the validity context.

5.3.1 Hypothesis

The hypothesis expresses the administrator’s belief about the behavior of the monitored system. The template of a hypothesis is given in Listing 5.3. The hypothesis is identified by a unique name; this allows the hypothesis to be saved in a database and later retrieved for future querying. The hypothesis describes a relationship (defined as the relation) between metrics (selected from a metric set) for some system configurations (defined as the context).

The relation, in turn, defines a mathematical function describing the relationship between metrics, a set of filters to process the monitoring data (e.g., remove noise), a method to find the best fit, and a mapping to calculate the confidence score from a relation specific goodness of fit. The relation is identified by a relation name and it may be used by several hypotheses. The relation is evaluated for each combination of metrics contained in a metric set. For example, the administrator may define that she expects the throughput-like metrics to be linearly related. In this case, the relation will be evaluated for each pair of metrics from a set of throughput-like metrics.

The hypothesis can also specify a validity range as a set of contexts over which the administrator expects the relationship to hold true. The context set is described using a set
Listing 5.3: SelfTalk Hypothesis Template. We show the template of a hypothesis defined in SelfTalk. It contains a name, describes a relationship between metrics in a validity context.

Listing 5.4: Definition of MySQL Query Throughput Metric. We show the SQL-like definition of a metric. The metric has several attributes that can be used to match the description given in the hypothesis.

of metric qualifiers similar to a metric set, but the description also specifies values defining the validity range. In the following, we describe each component of the hypothesis in detail.

**Metric:** The hypothesis describes a relationship between tuples of metrics, where each tuple is selected from a **metric set** also referred to as the **metric class**. The metric set, in turn, is constructed by a **join** of the available metrics (denoted by $M$). In more detail, a hypothesis may define a relationship between two metrics $x$ and $y$; then the metric set contains tuples of the form $(x_i, y_j)$, chosen from $M^2 = M \times M$, according to the join condition. In general, Dena supports metric sets of more than two metrics that are constructed from an expression evaluated on each metric’s attributes; the metrics that match the expression are included in the metric set.

The metric set is constructed from the description of metrics given with the hypothesis; Dena selects the metrics by matching the attributes to the conditions specified in the expression, similar to the SQL **JOIN** and a **WHERE** clause. The **metric** is a primitive entity that can be a performance measurement, a configuration setting, or a composite of several base performance metrics. Each metric has several attributes, such as, its name (e.g., `mysql_queries_per_sec`), the component name (e.g., MySQL) from where it is measured, the location of the component (e.g., hostname of the MySQL instance), and its unit of measurement (e.g., `query/sec` for throughput). For example, the measure of query throughput, `queries_per_sec` metric is defined in Listing 5.4, where MySQL is running on `cluster101`.
Constructing the metric set using expressions allows us to specify very broad qualifiers to capture a large set of metrics, or be very specific and capture metrics of a specific component. For example, we can express a relation between a broad set of throughput metrics, we specify the qualifiers as `x.unit = '1/sec' and y.unit = '1/sec'`, or we can express the metrics of a specific cache by specifying `x.name = 'cache_hits' and y.name = 'cache_gets' and x.location = y.location`.

The attributes of a metric are optional (except `id` and `name`) and the metric can be thought of as a schema-less relation. We use only the specified metric attributes to check a metric for inclusion into the metric set. In some cases, it is useful to define a composite metric built from a combination of several primitive metrics. The composite metric may be defined persistently within the Dena system, or temporarily by inlining the definition with the hypothesis. For example, for the cache, it is useful to define the cache miss-ratio as a composite metric that is computed as the ratio of number of cache accesses (`num_cache_gets`), and the number of cache misses (`num_cache_misses`).

**Relation:** The correlation between a set of metrics is described by a relation. The relation includes functions to filter the data, a mathematical function describing the relationship, an error function (e.g., squared error), a method to compute a best-fit (e.g., gradient descent), and a method to compute the confidence score. Many of these functions (e.g., the gradient descent optimizer and the method to compute the confidence score) are independent of the specific relation and may be shared by several relations.

As an illustration, consider the SelfTalk code snippet of the `linear` relation that is provided with the Dena runtime system. It shows the relation containing two parameters and two input data arrays. The parameters refer to the slope and y-intercept of the linear line and the two input arrays correspond to the input and output data values obtained by monitoring the system. We focus on the function to compute the confidence of the relation; the confidence score is a number between 0.0 and 1.0 representing how well the hypothesis fits the monitored data. In the example, we specify the confidence as the $R^2$, but we also check the residuals before returning the confidence score. The details of how Dena processes hypotheses are presented in Section 5.4.

**Context:** The relationship between metrics is influenced by the workload and other system configuration settings – referred to as the context of the hypothesis. Therefore, simply fitting the expectations to all measured data would lead to false fits. Consider the expectation `EXPECT LINEAR ('name= queries_per_sec', 'name= io_per_sec')` and assume that we get a 50% hit ratio with a 512MB cache and a 90% hit ratio with a 1GB cache. With
1 DEFINE RELATION linear {
2 PARAMETER a,b : number,
3 INPUT x:number−array, y:number−array,
4 ...
5 FUNCTION confidence
6 {
7 OUTPUT confidence:number
8 LANGUAGE 'matlab'
9 SCRIPT
10 $
11 y_{\text{hat}} = a \cdot x + b;
12 confidence = R^2(y, y_{\text{hat}});
13 //calculate residuals
14 ...
15 $
16 }
17 ...
18 }

Listing 5.5: Code Snippet of the Linear Relation. We show an example of how the confidence score is calculated for the linear relation. The snippet (written in MATLAB) shows the confidence score calculated as $R^2$ (coefficient of determination). We perform secondary checks to verify that there is no systematic deviation in the monitoring data.

different cache sizes, the exact relationship between the metrics (‘queries_per_sec’, ‘io_per_sec’) will be different. In fact, the factor $\alpha$ would be 0.5 for a 512MB cache and 0.9 for a 1GB cache. Thus, the administrator must provide her belief about the contexts that affect the validity of the hypothesis.

A context is expressed simply as a list of conditions on a set of performance metrics, workload metrics, or configuration parameters. In Listing 5.2, the context is specified as `name=cache_size and value<=512`, which states that the administrator expects the hypothesis to hold true only when the cache size is less than 512MB. We also support a wild-card operator, e.g., `name=cache_size and value=*`, that indicates that `cache_size` is a configuration parameter that may affect the fit; in this case, Dena will evaluate the expectation for each setting of the configuration separately.

5.3.2 Query

Dena expands the hypothesis submitted by the administrator into expectations, fits each expectation to the monitored data, and stores the results in a database. These results can be further analyzed by submitting queries written in SelfTalk. The administrator can query about the confidence of the expectations that result from the expansion of the hypotheses,
Listing 5.6: SelfTalk Query Template. The administrator can issue SelfTalk queries to analyze the results for different sub-components and contexts, and format the results grouping them by different attributes.

evaluate the fit under various contexts and for different sub-components. We categorize the queries into two types: (i) queries that focus the analysis on particular components, configurations, or confidence values, and (ii) queries that modify the presentation of the results by ordering them based on confidence score, or grouping them by particular metrics, or by grouping them by the configuration type.

The general syntax of a SelfTalk query is shown in Listing 5.6. It consists of three parts: (i) the preamble – we need to specify the name of the hypothesis being queried, e.g., the hypothesis name (shown in line 1), (ii) the query focus – we can narrow the analysis by specifying conditions on the metric set, the context set, and the confidence score (shown in lines 2-5) and (iii) the presentation of results – the results may be displayed in different order based on the confidence score, and may be grouped by various metric attributes or contexts (shown in lines 6-9). In the following, we present the details of how queries enable analysis of the results using two examples.

All queries include a hypothesis name. The hypothesis name is used to find the results stored by Dena in the database. If no options are specified, the results of all expectations that are generated from the hypothesis are returned — that is, the results of all possible expansions (expectations) of the metric set and context set declared in the hypothesis. This is equivalent to the SELECT * construct in SQL but SelfTalk allows the analysis to be done at a finer granularity with ease, by focusing the analysis to certain sub-components and on certain contexts. For example, Listing 5.7 returns results from expectations of the linear hypothesis (named HYP-LINEAR) for throughput-like metrics measured at the Akash storage server and MySQL only for configurations where the size of the MySQL cache is configured to 512MB and those expectations with a confidence score greater than 0.9.
Query to Return Results from MySQL and Akash. We can focus the results of the linear hypothesis by limiting the scope of the analysis to the MySQL throughput metrics and the throughput metrics from Akash storage server. In addition, the above query limits the analysis to those configurations where the size of the MySQL buffer pool is 512MB.

SelfTalk Query to Group by the Size of the MySQL Cache. The query returns the analysis of results between MySQL and Akash grouped by the MySQL buffer pool size. It provides the confidence scores per size as well.

In addition to allowing focused analysis of the results, SelfTalk allows the administrator to control the presentation of the results of a query by grouping, ordering, and ranking. As shown in Listing 5.8, we can analyze the effect of changing the size of the MySQL cache on the throughput by asking Dena to return expectations from the execution of the linear hypothesis for throughput-like metrics collected from MySQL and the Akash storage server, grouped by MySQL cache configurations (where the confidence scores are computed as the average for each cache configuration) and sorted by the confidence score in descending order.

5.4 Evaluating Hypothesis using the Dena Runtime System

Dena expands the hypothesis posed by the administrator to generate a larger set of expectations by enumerating all possible metrics and configurations that match the hypothesis. In this section, we describe the steps taken by Dena to validate each expectation with the monitored data and compute the confidence score.
5.4.1 Overview

An expectation is validated by evaluating how well the relationship described by the hypothesis applies to the monitored data. At its core, we apply statistical regression techniques to fit a function (describing the relationship between metrics) and evaluate the goodness of fit. While statistical regression techniques have been studied in great detail elsewhere [85], three main challenges exist in the implementation of a generic regression engine. We need to (1) process monitoring data collected from many different sources, (2) evaluate various relationships on the monitored data, and (3) compute a mapping from the relationship specific goodness of fit to a human-understandable confidence score.

The first challenge arises from the fact that monitoring data from a component contains noise and that monitored values from multiple components may not be aligned in time. Thus, we first filter the data to make it suitable for statistical regression. Filtering removes the outliers in the collected data and aligns the time-series data. After filtering, we can evaluate if the relation matches the monitored data. The second challenge is that the statistical regression techniques differ for different types of relations. While at the heart of all expectations is a mathematical function describing a relationship between a set of monitored metrics, the method of fitting the function differs from closed-form solutions (e.g., for linear regression) to iterative methods such as gradient descent. Finally, we need to compute a confidence score – a human understandable output between 0.0 (low confidence) and 1.0 (high confidence) from the relation-specific goodness metric. To aid in the design of a generic engine, we evaluate a set of commonly asked set of administrator’s questions and build a taxonomy of relations. In the following, we describe the taxonomy of relations and describe each of the steps in more detail. Then, we provide a list of sample relations used to evaluate the behavior of a multi-tier system.

5.4.2 Taxonomy

A relation describes a mapping between several metrics. Each relation specifies a function \( \hat{y} = \hat{f}(x) \) that describes how two metrics \( x \) and \( y \) are expected to be correlated. The relationship may be comparisons – where the mapping between \( x \) and \( y \) is a boolean operator e.g., \( y < x \) or regressions – where the mapping between \( x \) and \( y \) is a mathematical function e.g., \( y = ax + b \). In addition, each of the relationships may be time dependent e.g., \( \hat{y}_t = \hat{f}(x_t) \) or time-independent.
We classify the relations into different categories using the above criteria as shown in Figure 5.1. The relations are first classified into two categories: *regressions* and *comparisons*. The relations classified into *regressions* are functions that describe a mathematical relationship between several metrics. An example of a *regression* relationship is a linear relationship between two metrics; the function mapping $x$ to $y$ is described by $\hat{y}_{\alpha, \beta}(x) = \alpha x + \beta$. The validity of these relations can be evaluated using statistical regression techniques. The second relation type is a *comparison* where the mapping between two metrics is a boolean operator ($<, >, =, \leq, \geq$). In this case, directly applying statistical regression techniques is difficult. Thus, we evaluate the validity of these relations using simple counting where we validate the relation by counting the fraction of points where the comparison holds true.

Hypotheses from each of the above two categories (*regressions* and *comparisons*) can be applied to time-dependent or time-independent data. Time-dependent relations treat the input as a time-series where the relation between input metrics are considered through time – that is, the input data to the relation is tuples of metric values with same time-stamps. On the other hand, time-independent relations treat the data as an unordered list. We explain the details next.
5.4.3 Evaluating Expectations

The evaluation of an expectation consists of three steps: (1) collect and filter monitored data, (2) apply statistical regression and evaluate for monitored data, and (3) compute the confidence score.

Step 1 – Filtering Monitored Data: The monitoring data collected from components have two sources of error: (1) noise in the data collected from one component, and (2) misalignment of data collected from multiple components. We filter the data values before evaluating the relationship.

The noise in the monitored data is seen as outliers in the data. The outliers occur when data is collected from components during their initialization phases either at startup or after a configuration change, and due to interference from background tasks. One such example is the measurement of the num_cache_hits (the number of cache hits) and num_cache_misses (the number of cache misses) from a cache. During the initialization phase (i.e., cache warm-up), the cache misses are high as many of cache accesses experience cold misses since the cache is empty. However, as the cache warms up, the number of cache misses reduces steadily (conversely the number of cache hits increases steadily) until the values reach steady state. Similarly, infrequent background tasks from the operating system or transient network bottlenecks introduce noise in the measurements as well. We filter these outliers before applying statistical regression. We choose to use percentile filtering due to its simplicity. Percentile filters are generic; they make no assumptions about the distribution of data other than that the number of samples is large enough to cover most regions of the underlying distribution. We use percentile filtering where the top $t\%$ and the bottom $b\%$ (typically 5%) of sampled data is removed. By removing these samples, the percentile filter keeps the samples which form the majority in the distribution. The filtering process is different for time-independent relations; in these, we perform percentile filtering per configuration value rather than on the entire dataset.

Time-series data pose an additional challenge where the data measured at different components may be misaligned due to clock skew as well as due to causality between components. We evaluate time-series data by matching (i.e., joining in the database terminology) the sampled values using the timestamp. Causality between components can also account for some misalignment between the sampled metrics. For example, a change in the workload is reflected at the metrics collected at the higher layers (e.g., the database) before it is seen in the metrics collected in the lower layers (e.g., disk). While there are various sophisticated
methods for aligning time-series data, we find simple techniques of grouping values using a
coarser-time granularity and using moving average filters work well; for example, we align
the data values by grouping them into a coarse timestamp granularity (e.g., 10 seconds). We
also use a moving-average filter. A moving average is used to analyze a set of data points
by creating a series of averages of adjacent subsets of the full data set; this smooths out
short-term fluctuations while maintaining the long-term trends. Aligning time-series data
by estimating the clock skew and delay between components is an area for improvement; we
leave this optimization as future work.

**Step 2 – Performing Regression:** After filtering the monitored data, we perform
statistical regression to evaluate how well the hypothesis fits the measured values. We find
the best values of the *free parameters* to reduce the squared error between the hypothesis
and the measured values. For example, consider the linear relation,

\[
\hat{y}_{\alpha,\beta}(x_t) = \alpha x_t + \beta
\]  

with two free parameters \(\alpha\) – the slope of the line, and \(\beta\) – the y-intercept of the line. The
best fit of the relation to the measured data is obtained when the squared-error between the
predicted values and the measured values is minimized. We define the error (i.e., how the
relation deviates from the measured values) as

\[
\xi(\alpha, \beta) = \sum_{(x,y)} [(y - \hat{y}_{\alpha,\beta}(x))^2]
\]

and we find the best-fit of the relation by mapping the problem of reducing the squared
error as an optimization problem and use standard optimization techniques such a gradient
descent (using the partial derivatives if given) to find the best parameter values. In some
cases, the best parameter values can be obtained from closed-form solutions (such as for
linear regression). We opt for the closed-form solution rather than iterative search in these
cases.

**Step 3 – Computing the Confidence Score:** After applying statistical regression
and optimizing the free parameters, we evaluate how well the relation describes the data
and report the *confidence score*. The *confidence score* is a human-understandable number
between 0.0 and 1.0, which indicates a poor and good fit respectively. The evaluation of
the confidence score is dependent on the relation — specifically, whether the relation is a
*comparison* or *regression*. 
For the comparison relations, the confidence score is the fraction of data when it holds true; we count the number of times the comparison evaluates to true and divide by the total number of monitored data points. For regression functions (i.e., those with a mathematical relationship), we use the coefficient of determination, $R^2$, to compute the confidence score. The $R^2$ is a fraction between 0.0 and 1.0. A $R^2$ value of 0.0 indicates that the function does not explain the relationship between the two metrics. Assuming a relation is defined as $\hat{y} = \hat{f}(x)$, the coefficient of determination is defined as

\[ R^2 = 1 - \frac{SS_{err}}{SS_{tot}} \] (5.3)

\[ SS_{err} = \sum_i (y_i - \hat{f}(x_i))^2 \] (5.4)

\[ SS_{tot} = \sum_i (y_i - \bar{y})^2 \] (5.5)

where $SS_{err}$ is the residual sum of the squares, $SS_{tot}$ is the total sum of squares, and $\bar{y} = mean(y)$. However, simply using $R^2$ to evaluate the fit may be incorrect. To better evaluate the fit, we perform a secondary test using the residuals of the regression; the residuals are the vertical distances from each point in the fitted line to the monitored data. A good fit has the residuals equally above and below the fitted line. If the residuals are not randomly scattered – indicating a systematic deviation from the fitted line then, the $R^2$ value may be misleading. In this case, we report that the fit has a low confidence score.

### 5.5 Validating Performance of a Multi-tier Storage System

In this section, we provide a sample of hypotheses that we issue to understand and validate the behavior of a multi-tier storage system consisting of a MySQL database using a virtual volume hosted on the Akash storage server. We choose one or two hypotheses from each of the categories we describe in the relation taxonomy. For each hypothesis, we provide the high-level question the administrator is probing, the underlying regression/comparison function tested in the hypothesis, the filtering applied to the monitored data, and the optimization algorithm used to find the best fit.

#### 5.5.1 Time-dependent Regression

The LINEAR hypothesis is one of the simplest hypothesis that an administrator can issue to Dena. We issue this hypothesis to diagnose traffic patterns along the storage path.
Specifically, as an administrator, we ask the question – “I expect the throughput measured at the storage system to be linearly correlated with the throughput measured at MySQL” or more generally “I expect the throughput metrics along the storage path to be linearly related” with the belief that as we increase the load at the MySQL database, the load on the underlying storage server will increase correspondingly. The linear relation is defined as

\[ \hat{y}_{\alpha,\beta}(x_t) = \alpha x_t + \beta \]  

(5.6)

with two free parameters: \( \alpha \) and \( \beta \). We filter the time-series data by first removing the outliers using percentile filtering and then smooth the values with a moving average filter. The line is fit to the monitored data using linear regression and we use the coefficient of determination \( (R^2) \) as the confidence score. We further verify the fit using the residuals to determine if the data does not systematically deviate from the hypothesis. If the residuals are not valid, we report that the hypothesis is not a good fit.

In addition, Dena can incorporate results from models, such as those derived from operational laws, to verify the behavior of a multi-tier system; an example of this is the LITTLE hypothesis that defines a relationship between throughput and latency using Little’s law [46]. Little’s law states that if the system is stable then, the response time and throughput are inversely related. We issue this hypothesis to verify that the behavior of the system adheres to the behavior explained by operational laws; a stable system follows these laws. For example, the administrator can express her belief in the operational laws by making a high-level hypothesis that “I expect the throughput measured at the storage system is inversely correlated with the latency measured at MySQL”. For an interactive system, such as multi-tier storage systems, Little’s law is expressed as

\[ \hat{X}_{N,Z}(R_t) = \frac{N}{R_t + Z} \]  

(5.7)

with two free parameters: \( N \) and \( Z \), which are number of clients and average think time respectively, and \( X_t \) and \( R_t \) denoting throughput and response time. Similar to the processing of linear relation, we filter the data by first removing the outliers using percentile filtering and then smoothing the values with a moving average filter. The curve is fit to the monitored data using gradient descent optimization, and we use the coefficient of determination \( (R^2) \) as the confidence score. We further verify the fit using the residuals to determine if the data does not systematically deviate from the hypothesis; if the residuals are not valid, we report that the hypothesis is not a good fit.
5.5.2 Time-independent Regression

Our storage system uses the quanta-based scheduler to divide the storage bandwidth among several virtual volumes. The quanta-based scheduler partitions the bandwidth by allocating a time quantum where one of the workload obtains exclusive access to the underlying disk. For modeling the quanta latency, we observe that the typical server system is an interactive, closed-loop system. This means that, even if incoming load may vary over time, at any given point in time, the rate of serviced requests is roughly equal to the incoming request rate. Then, according to the interactive response time law [46]:

\[ L_d = \frac{N}{X} - Z \]  

(5.8)

where \( L_d \) is the response time of the storage server, including both I/O request scheduling and the disk access latency, \( N \) is the number of application threads, \( X \) is the throughput, and \( Z \) is the think time of each application thread issuing requests to the disk. We then use this formula to derive the average disk access latency for each application, when given a certain fraction of the disk bandwidth. We assume that think time per thread is negligible compared to request processing time, i.e., we assume that I/O requests are arriving relatively frequently, and disk access time is significant. Then, through a simple derivation, we arrive at the following formula

\[ L_d(\rho_d) = \frac{L_d(1)}{\rho_d} \]  

(5.9)

where \( L_d(1) \) is the baseline disk latency for an application, when the entire disk bandwidth is allocated to that application. This formula is intuitive. For example, if the entire disk was given to the application, i.e., \( \rho_d = 1 \), then the storage access latency is equal to the underlying disk access latency. On the other hand, if the application is given a small fraction of the disk bandwidth, i.e., \( \rho_d \approx 0 \), then the storage access latency is very high (approaches \( \infty \)). The QUANTA hypothesis expresses the above belief from the operational law model where we expect the storage access latency of the application to be inversely related to the allocation time fraction. The QUANTA hypothesis uses the inverse relationship that is described as

\[ \hat{y}_{\alpha,\beta}(x) = \frac{\alpha}{x^\beta} \]  

(5.10)

where the waiting time at the scheduler (\( y \)) is inversely related with the time fraction (\( x \)) given to the application. We filter the latency values using the percentile filter and average
the samples (for each quanta setting) before performing regression. We find the best-fit for
the free parameters using gradient descent and we use $R^2$ as the confidence score and use
the residuals as secondary check.

5.5.3 Time-dependent Comparison

The LESS/EQ hypothesis is used to answer many storage questions. For example, the admin-
istrator can check a configuration parameter — “I expect the current size of the cache is less
than or equal to the maximum size (as defined in the configuration)” or check a performance
metric — “I expect the latency (e.g., response time) measured at higher level components
(MySQL) is higher than the latency measured at the lower level components (disk)”’. We
remove the outliers using percentile filtering and use a moving average filter to synchronize
the samples over time. There is no regression step and we report the confidence score as the
fraction of samples where the comparison ($\leq$) holds true.

5.5.4 Time-independent Comparison

The miss-ratio curve (\textsc{MRC}) relation describes the behavior of a cache; it states that the cache
miss-ratio (i.e., the ratio of cache misses to the cache accesses) is a monotonically decreasing
curve with respect to the cache size. We capture this relationship in two ways: by comparing
to a user-provided miss-ratio function or systematically checking that the curve is indeed
monotonically decreasing.

In the first method, the administrator may provide the expected miss-ratio curve from
a model (i.e., using Mattson’s stack algorithm [67]) or from a cache simulator; with either
approach, we are given a list of tuples of the form $\langle c, m \rangle$ (where $c$ is the cache size and $m$
the miss-ratio) and we evaluate the confidence using $R^2$.

In the second method, for each cache size $c$, we obtain the values of the miss-ratio and
apply the percentile filter; the filtering concentrates the miss-ratio samples into a cluster
(for each cache size $c$). Then, we average the miss-ratios and use the resulting list $\langle c, \bar{m} \rangle$
of tuples (where $\bar{m}$ is the average of the miss-ratios for cache size $c$) to sort by cache sizes
in ascending order and verify that the miss-ratio keeps decreasing (or remains flat) as the
cache size is increased. We count the fraction of times the comparison holds true and report
it as the confidence score.

The \textsc{Const} hypothesis checks if the values of a metric are constant; we use this relation
to issue hypothesis of the form “I expect that the size of the cache (i.e., the number of items
Table 5.1: **Programming Effort.** We show the programming effort needed to implement the different components of SelfTalk/Dena. The code consists of Java-based components to communicate with the underlying DBMS system storing the monitoring data and mathematical functions written in MATLAB to process the hypothesis.

The size in the cache remains constant”. We note that there is a small fraction of time (during start-up) when the size is not equal to the capacity which is filtered by the percentile filter. We filter the data using percentile filter to remove outliers and return high confidence if samples are almost constant – that is, the variation in the values is within a small ratio of its mean; we compute the ratio of the mean of \( x \) and divide by the standard deviation. If the ratio is less than a threshold, we report a confidence score of 1, otherwise we report a confidence score of 0.

### 5.6 Prototype Implementation

The *Dena* runtime system is composed of multiple parts: a front-end consisting of the SelfTalk parser, a core regression engine, and a database backend storing the monitoring data. The monitoring data is collected from existing software; we use built-in instrumentation such as the MySQL/InnoDB monitor to get statistics from the database, `vmstat` and `iostat` to obtain statistics from the operating system, and built-in instrumentation from our Akash storage server (described in Chapter 3). We implement the core of the statistical regression algorithms using MATLAB utilizing JDBC to fetch data from the backend database (PostgreSQL). We provide simple relations that can be utilized by a novice administrator. This includes all the relations we describe in Section 5.5 plus we provide relations describing exponential and polynomial curves, and all boolean comparisons.

The administrator can specify the hypothesis at the command-line or by referring *Dena* to a file. Given a hypothesis, *Dena* parses the details and expands the hypothesis to all possible expectations. *Dena* instantiates a new object for each expectation, obtains the data from the database, fits the relation to the monitored data, and computes the confidence score. When the fitting is complete, the details of the hypothesis, the set of expectations, the final
fitted values of the free parameters, and the descriptions of the contexts are stored into the
database for future analysis by the administrator. We summarize the effort required to build
the various parts as lines of code in each component in Table 5.1.

5.7 Evaluation Methodology

We use three workloads: a simple micro-benchmark, called Workload-C, and two industry-
standard benchmarks, TPC-W and TPC-C. The workloads are described in Section 4.10.2. We
run our Web based applications (TPC-W) on a dynamic content infrastructure consisting of
the Apache web server, the PHP application server and the MySQL/InnoDB (version 5.0.24)
database engine. We run the Apache Web server and MySQL on Dell PowerEdge SC1450
with dual Intel Xeon processors running at 3.0 Ghz with 2GB of memory. MySQL connects
to the raw device hosted by the Akash server. We run the Akash storage server on a Dell
PowerEdge PE1950 with 8 Intel Xeon processors running at 2.8 Ghz with 3GB of memory.
To maximize I/O bandwidth, we use RAID-0 on 15 10K RPM 250GB hard disks. Non-web
application utilize the same MySQL and storage server instances; however, they do not use
the machine running the Apache web server.

The monitoring data is collected from the underlying operating system (using Linux
utilities vmstat and iostat), the MySQL database, and the Akash storage server. The
collected metrics are timestamped using gettimeofday(). The metrics are collected over a
period of 11 months resulting in approximately 50 gigabytes of data in the database.

5.8 Results

We evaluate the efficacy of Den a to validate the overall system behavior and to understand
the per-component behavior. First, we issue broad high-level hypotheses describing the rela-
tionships in a multi-tier storage system and check the validity of these relationships. Second,
we issue specific queries to provide insights into the behavior of a specific component and
also one component’s effect on other components within the multi-tier system. Third, we
present additional results studying cases where there is a mismatch between the administra-
tor’s belief and the monitoring data. Finally, we present measurements calculating the cost
and time breakdown of executing a hypothesis.
<table>
<thead>
<tr>
<th>Hypothesis</th>
<th>Expectations</th>
<th>Avg. Confidence</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINEAR</td>
<td>3072</td>
<td>86%</td>
</tr>
<tr>
<td>LESS/EQ</td>
<td>3488</td>
<td>98%</td>
</tr>
<tr>
<td>LITTLE</td>
<td>3290</td>
<td>92%</td>
</tr>
</tbody>
</table>

Table 5.2: **Expectations.** We show the number of expectations generated for each high-level hypothesis.

### 5.8.1 Understanding the Behavior of the Overall System

We issue several broad high-level hypothesis to check the overall behavior of the system. We present the correlations that *Dena* discovers for three simple hypotheses: (1) **LINEAR** – expects that metrics of the same type are linearly correlated, (2) **LESS/EQ** – states that round-trip latency is additive across layers and (3) **LITTLE** – states that throughput and latency adhere to the Little’s law. Table 5.2 shows the number of expectations generated for each hypothesis for all contexts. *Dena* generates the expectations automatically for a given hypothesis. Figure 5.2 shows the correlations discovered by *Dena* in a graph where the *nodes* represent metrics and the *edges* indicate a correlation. To simplify the presentation, we only show metrics related to the throughput and latency for each module. In addition, we only show results where we configure the cache to 1 GB resulting in a 50% miss-ratio and allocate the entire disk bandwidth to the application. We explain the correlations discovered for the **LESS/EQ** and **LITTLE** in detail next.

For the **LINEAR** hypothesis, shown in Figure 5.2a, we find two clusters of metrics: a set of throughput related metrics and a set of latency related metrics. First, we see that the set of throughput metrics is linearly correlated. This is expected because the storage is configured as a single path from the NBD module to the disk module. The cache and quanta modules do not affect the linear correlation between the throughput seen in the NBD module (*nbd_enter*) and the disk module (*disk_enter*) because while the cache causes less I/Os to be issued to disk, an increase in the rate of I/O requests entering the storage system still results in a corresponding increase in the rate of disk I/Os. Similarly, latency across components is linearly correlated as well except the quanta module; it controls the number of requests entering disk leading to an additional queuing delay between the *disk_latency* and the *quanta_latency* breaking the linear relationship across latencies [98].

We develop the **LESS/EQ** hypothesis by using the information of the structure of *Akash* which allows us to hypothesize that latencies (and similarly throughput) measured in some modules are less than the latencies measured in other modules. Figure 5.2b shows our
results using a directed graph where the arrowhead points from the smaller metric to the larger metric. For example, the cache module sits above the quanta module and forwards requests only on cache misses. Therefore, with a 50% miss-ratio, the latency at the cache module is less than the quanta module. This is shown as an arrow from cache_latency to quanta_latency. Conversely, the number of requests entering the quanta module is less than the number of requests entering the cache module, shown as an arrow from quanta_enter to cache_enter.

As Akash is closed-loop storage system, we hypothesize that performance adheres to Little’s law [46] — that is, the throughput and latency metrics follow the interactive response time law and thus are inversely proportional. Figure 5.2c shows that indeed the system complies with Little’s law as the throughput and latency metrics are indeed correlated. The disk_latency is not correlated with Little’s law as the quanta module self-adjusts its scheduling policy to varying disk service times [98] leading to a weak correlation with the disk_latency.

Figure 5.2: **Correlations.** We show the pairwise correlations we discover for different administrator hypotheses in the above graph. The nodes represent different metrics and the edges show the correlation. The above results were gathered with a 1GB cache resulting in a miss-ratio of 50%, and the entire disk bandwidth was allocated to the application.
5.8.2 Understanding Per-Component Behavior

Next, we explore the behavior of different storage server components by studying the correlations found using different hypotheses. We focus on the two major components: the cache and the quanta-based I/O scheduler modules within Akash. Then, we present results showing how Dena can be used to study interactions between multiple components as well. To illustrate this, we focus on the effect of cache inclusiveness in multi-tier caches.

Understanding the Cache: We study the effect of caching on the performance of the storage system by issuing several hypotheses that provide an insight into its behavior: MRC – indicates the administrator’s belief that the cache performance will improve (i.e., its miss-ratio will decrease) as the size of cache is increased, LESS/EQ – states that caching improves performance by reducing latency where the latency to access items from the cache is lower than the latency of accessing items from the underlying disk, and LINEAR – states the belief that since the cache size impacts performance, the linear relation between metrics must account for the size of the cache as a context. We evaluate these beliefs using the Workload-C workload which has a miss-ratio of 75% with a small cache (256MB), 50% with a medium cache (512MB) and a 12% with a large cache (896MB). Figure 5.4 shows the results of the MRC and LINEAR hypothesis.

Figure 5.3, shows the miss-ratio for the Workload-C workload. As expected, the miss-ratio is monotonically decreasing – a straight line from approximately 1.0 (many misses) with a small cache to near 0.0 (many hits). Dena computes a confidence score of 0.99 for the miss-ratio curve. Regardless of the cache size, caching provides a benefit in terms of performance. This improvement can be checked using the LESS/EQ hypothesis; Dena reports a confidence score of 1.0 for all cache sizes indicating that the throughput measured at the cache is higher than the throughput at the underlying disk and the latency at the cache is lower than the latency of fetching data from disk.

The detailed impact on the performance from different cache sizes can be obtained by issuing the LINEAR hypothesis as seen in Figures 5.4a and 5.4b. Each plot shows three lines corresponding to three cache sizes: a small cache (shown in red with squares), a medium cache (shown in green with triangles), and a large cache (shown in blue with circles). The points are the samples (after percentile filtering) obtained through monitoring and the line is the best-fit of the relation described in the hypothesis. The plots show that performance can indeed be improved by increasing the size of the cache; the throughput ratio between the cache and the disk (i.e., the factor of improvement) is 1.25, 2, and 8 for small, medium,
Figure 5.3: **Understanding the Cache Behavior.** We look at the impact of caching on the performance of the storage server by studying the miss-ratio curve and comparing the the throughput and latency across the cache module within *Akash*.

Figure 5.4: **Impact of Caching on Performance.** We look at the impact of caching on the performance of the storage server by studying the miss-ratio curve and comparing the the throughput and latency across the cache module within *Akash*.

and large cache sizes respectively. Similar factors are seen in the reduction of the access latency at the cache and the underlying disk latency.

**Understanding the Quanta Scheduler:** The quanta scheduler is the mechanism *Akash* uses to proportionally allocate the disk bandwidth among multiple storage clients. As we describe in Section 5.5, the effect on performance can be modeled using operational laws. In this case, we observe that the *Akash* is a closed-loop system where the rate of serviced requests is roughly equal to the incoming request rate. Then, by using the *interactive*
Figure 5.5: Understanding the Quanta Behavior. We see that the impact of the quanta scheduler is inverse where halving the disk bandwidth fraction leads to a doubling of the quanta latency.

Using the response-time law, we derive the relationship that the latency as seen from the quanta module varies inversely with fraction of the disk bandwidth allocated to the workload – that is, as the fraction of disk bandwidth is halved, the per-request latency doubles.

Figure 5.5 presents the results obtained from Dena for the Workload-C workload. It shows three curves showing the results for the small, medium, and large cache sizes. In addition, we plot the measured values of the quanta latency for comparison. The results show that our belief that the latency varies inversely to the disk bandwidth fraction is correct; the fitted curve closely matches the observed values resulting in confidence scores of 0.94, 0.94, and 0.93 for the small/medium/large caches respectively. Using the QUANTA hypothesis allows us to understand the disk performance as well. Specifically, Dena shows that the confidence score for the large cache is slightly smaller than the confidence scores for small and medium cache sizes. The reason is that there is a higher variability of the average disk latency when (i) the underlying disk bandwidth isolation is less effective due to frequent switching between workloads and (ii) disk scheduling optimizations are less effective and reliable due to fewer requests in the scheduler queue. However, even with this variability, the underlying relationship is still inverse leading Dena to report a high confidence score.

Understanding Two-tiers of Caches: In a multi-level cache hierarchy using the standard (uncoordinated) LRU replacement policy at all levels, any cache miss from cache level \( q_i \) will result in bringing the needed block into all lower levels of the cache hierarchy, before providing the requested block to cache \( i \). It follows that the block is redundantly
Figure 5.6: **Understanding the Two-tier Cache Behavior.** We see the effect of cache inclusiveness in the miss-ratio at the second-level cache. The miss-ratio increases steadily as the size of the first-level cache is increased.

cached at all cache levels, which is called the *inclusiveness* property [111]. Therefore, if an application is given a certain cache quota $\rho_i$ at a level of cache $i$, any cache quotas $\rho_j$ given at any lower level of cache $j$, with $\rho_j < \rho_i$ will be mostly wasteful. We can verify this behavior using two hypothesis based on the MRC hypothesis. Due to cache inclusiveness, the administrator expects that by increasing the size of the first-level cache (i.e., the MySQL buffer pool) the performance of the second-level cache (i.e., the miss-ratio at the storage server cache) steadily decreases due to lower temporal locality.

We perform the analysis by stating that the relationship between the miss-ratio at the storage cache and the size of the MySQL buffer pool size is monotonically increasing; the context of the hypothesis is the storage cache size. Given this hypothesis, *Diena* presents these results grouped by each storage cache size. We present the results graphically for the TPC-W workload. Figure 5.6 shows this behavior for three different storage cache sizes: small (128MB), medium (512MB), and large (896MB) where the lines indicate the best-fit regression and the points are measured values. For the small storage cache (shown in blue), we see that the miss-ratio is high at 80% for small MySQL buffer pool sizes but quickly increases to 100% for medium to large MySQL buffer pool sizes. For a large storage cache (shown in red), the effect is more clear; the miss-ratio for a small MySQL cache is less than 25% but the miss-ratio worsens steadily as the MySQL cache is increased where it crosses 50% after 512MB of MySQL buffer pool and over 90% for very large MySQL cache sizes.
Figure 5.7: **Different Errors.** *Dena* does not expect the administrator to issue correct hypotheses or the system to behave correctly. In both cases, there is a mismatch between the administrator and the system leading to low confidence scores. We show three such scenarios where: (i) the system is faulty, (ii) hypothesis is faulty, and (iii) the context is faulty.
5.8.3 Understanding Mismatches

There can be a mismatch between the administrator’s beliefs and the monitored data. This can occur either due to a fault in the system or from a misunderstanding of the system by the administrator. In either case, Dena reports low confidence scores and the administrator may probe deeper by issuing different hypotheses to diagnose faults or to improve her understanding of the system. In the following, we present three cases of mismatch. We test for cases where (i) the system is faulty – we induce a fault in the cache resulting in errors in the cache replacement policy, (ii) the hypothesis is faulty – we hypothesize that the behavior of the quanta scheduler is linear, and (iii) the context is faulty – we hypothesize the metrics of the same type are linearly correlated but fail to provide the context information that the size of the storage cache may affect the relationship.

System is Faulty: In the first case, we show results showing how Dena can be used to detect a fault in the system. We detect a fault in the cache replacement policy using the MRC hypothesis which states that “I expect the cache misses to decrease monotonically with increasing cache size”. We run the Workload-C workload for this experiment. In an earlier case, we have shown that the Workload-C workload has a straight line as the miss-ratio curve, shown in Figure 5.3, and that with a fault-free cache replacement algorithm, the curve is indeed monotonically decreasing. Now we induce a fault in the cache replacement algorithm that reduces caching benefit. It has more cache misses than expected for some cache sizes as shown in Figure 5.7a. Due to the fault, Dena is not able to validate the relationship using the monitoring data leading Dena to report a very low confidence score of 0.24. This scenario highlights one use-case where the administrator is confident in her hypothesis and thus can conclude that the system is faulty.

Hypothesis is Faulty: Another case where there is a mismatch between the administrator and the system is if the administrator’s belief is incorrect. We test a case by issuing the hypothesis that we (falsely) expect the “latency of the quanta module is linearly related with the disk bandwidth fraction”. During the design phase of Akash, we made a similar assumption when we noticed that the throughput of the storage system varies linearly with the disk bandwidth fraction (by applying Little’s law) and incorrectly concluded that the effect on latency is linear as well. We have shown that the relationship is indeed inverse earlier in Figure 5.5. The error is noticed by Dena, as shown in Figure 5.7b, where the expected line does not match the monitored data. In this case, Dena reports a confidence score of 0.8. This scenario describes the second use-case where the administrator initiates
Figure 5.8: **Timing Hypothesis Execution.** We measure the time to execute an expectation and notice that the bulk of the cost is fetching the data from the database while the time needed to perform statistical regression is small.

a dialogue to understand the behavior of the system by issuing hypotheses (correctly or incorrectly) and obtaining feedback on its validity.

**Context is Faulty:** In the last case, we re-issue the LINEAR hypothesis but fail to identify that the size of the cache may affect the validity of the hypothesis. With an incorrect context, the relation cannot be fit; as Figure 5.7c shows, the data values form several lines with different slopes and y-intercepts and no single line satisfies the monitored data. With an incorrect context, the best-fit of a line is a null fit and the confidence score is 0.0. A similar result for latency is shown in Figure 5.7d.

### 5.8.4 Cost of Hypothesis Execution

We also evaluate the cost of executing a hypothesis by measuring the time taken to fetch the data from the database and the time needed to perform statistical regression. To evaluate the cost, we run several hypotheses of varying computation costs. As Figure 5.8 shows, the computation time is a fraction of the time needed to fetch the data from the database. It takes roughly 1 to 1.5 seconds (average) to fetch the data for an expectation and less than 40ms to find the best-fit. The computation cost is the least for comparison relations — these perform simple counting thus require less than 5ms to report the confidence score. The regression cost is higher as we need to fit the line to the monitored data. Specifically, the time needed to find the closed-form solution for LINEAR is 25ms and the time needed for QUANTA (inverse) is 39ms on average.
5.9 Summary

We introduce *SelfTalk* – a declarative high-level language, and *Dena* – a novel runtime tool, that work in concert to allow users to interact with a running system, by hypothesizing about expected system behavior, and posing queries about the system status. Using the given hypothesis and monitoring data, *Dena* applies statistical models to evaluate whether the system complies with the user’s expectations. The degree of fit is reported to the user as confidence scores. *SelfTalk* and *Dena* thus provide the basis for evolving system self-expression in a self-managed system towards the human-like ability to agree or disagree with the system administrator on facts and beliefs about the system in relation to given environments/contexts. We evaluate our approach on a multi-tier dynamic content web server consisting of an Apache/PHP web server, a MySQL database using storage hosted by a virtual storage system called *Akash* and find that *Dena* can quickly validate user’s hypotheses and accurately diagnose system misbehavior.
Chapter 6

Related Work

The techniques presented in this dissertation build on previous research in several different areas: consolidated storage architectures, techniques for monitoring performance and diagnosing bottlenecks, techniques for improving performance, and techniques for improving sharing in storage systems. In this chapter, we describe the relevant work in each area.

6.1 Networked Storage Designs

The desire to reduce operational costs has driven modern datacenters towards server consolidation. The common technique to store the data of multiple applications on a physical device is by virtualizing the storage device. Virtualization introduces an indirection point such that the user or application sees the familiar interface, while carving the underlying hardware into many virtual devices. This technique has been applied to different storage systems layers from high-level database systems that use middleware to present a virtual database [4, 5, 96] to the low-level storage controllers that manage a set of disks [76]. In particular, we focus our study on previous work closely related to our own and present previous work in two categories of networked storage: (i) file-based systems – where the system presents a virtualized filesystem (ii) block-based systems – where the system provides a virtual block device and spreads the data over multiple devices.

Shared file-based systems evolved from previous research into distributed filesystems [43, 55]. In particular, the common design in local area networks (LANs) is to export the local filesystem using the NFS protocol and enable access to the filesystem from multiple machines – creating rudimentary file servers [55]. However, as these machines typically do nothing else than handle filesystem requests specialized file server appliances have been constructed.
as well [42]. Networked file servers combine a large set of disks with a specialized operating system to provide multiple virtual filesystems accessible through the NFS, CIFS, and HTTP protocols. By consolidating storage, the file servers can enable other useful features such as automatic backup, snapshots of the filesystem, and de-duplication of the copies of files.

Similar to file servers, block-based storage servers allow for storage consolidation. However, block-based systems expose a much simpler interface. They present a block device that can read/write data from different offsets with the virtual device. The block device interface is simpler to build while file-based systems need to support a myriad of filesystem operations. In addition, block-based servers allow many of the salient features present in file servers such as periodic backup, snapshots, and de-duplication of data. Recent examples of block-based storage servers include FAB [86], Kybos [108], and Ursa Minor [2]. Previous work in block-based storage servers focus on the design and implementation of these systems and not on policies to effectively utilize the available resources. In the course of our work, we design the Akash storage server that builds on the concepts presented in previous work. Specifically, Akash supports multiple storage volumes, provides partitioning of the cache and the disk bandwidth, and can be deployed over commodity storage hardware.

6.2 Caching and Prefetching

This section discusses related techniques for improving caching efficiency at the storage server, including: (i) collaborative approaches like our own, which pass explicit hints between client and storage caches, or require more extensive code restructuring and reorganization, (ii) gray-box approaches, which infer application patterns at the storage based on application semantics known \textit{a priori}, and (iii) black box approaches, which infer application patterns at the storage server in the absence of any semantic information.

Explicitly Collaborative Approaches

Several approaches pass explicit hints from the client cache to the storage cache [19, 34, 60, 77]. These hints can indicate, for example, the reason behind a write block request to storage [60], explicit demotions of blocks from the storage client to the server cache [111], or the relative importance of requested blocks [22]. These techniques modify the interface between the storage client and server, by requiring that an additional identifier (representing the hint) be passed to the storage server. Thus, similar to \textit{QuickMine}, these techniques improve storage cache efficiency through explicit context information. However, as opposed to
our work, inserting the appropriate hints needs a thorough understanding of the application internals. For example, Li et al. [60] require the understanding of database system internals to distinguish the context surrounding each block I/O request. In contrast, we use readily available information within the application about preexisting contexts.

In general, collaboration between the application and storage server has been extensively studied in the context of database systems, e.g., by providing the DBMS with more knowledge of the underlying storage characteristics [87], by providing application semantic knowledge to storage servers i.e., by mapping database relations to objects [81], or by offloading some tasks to the storage server [84]. Other recent approaches in this area take advantage of context information available to the database query optimizer [31, 40], or add new middleware components for exploiting explicit query dependencies e.g., by SQL code re-writing to group related queries together [12]. Unlike our technique, these explicitly collaborative approaches require substantial restructuring of the database system, code reorganization in the application, or modifications to the software stack in order to effectively leverage semantic contexts. In contrast, we show that substantial performance advantage can be obtained with minimal changes to existing software components and interfaces.

**Gray-box Approaches**

Transparent techniques for storage cache optimization leverage I/O meta-data, or application semantics known *a priori*. Meta-data based approaches include using file-system meta-data, i.e., distinguishing between i-node and data blocks explicitly, or using filenames [16, 56, 59, 92, 114], or indirectly by probing at the storage client [6, 7, 94]. Alternative techniques based on application semantics leverage the *program backtrace* [34], user information [114], or specific characteristics, such as in-memory addresses of I/O requests [23, 51] to classify or infer application patterns.

Sivathanu et al. [93] use minimally intrusive instrumentation to the DBMS and log snooping to record a number of statistics, such that the storage system can provide cache exclusiveness and reliability for the database system running on top. However, this technique is DBMS-specific, the storage server needs to be aware of the characteristics of the particular database system. In eviction-based prefetching [23], the storage cache detects whether a block has been evicted from the client cache by matching the in-memory address of a newly requested block with that of a block requested previously. The storage cache then issues prefetches for these blocks. Graph-based prefetching techniques based on discovering correlations among files in filesystems also fall into this category [36, 57], although they are
not scalable to the number of blocks typical in storage systems [61].

In contrast to the above approaches, our work is generally applicable to any type of storage client and application; any database and file-based application can benefit from QuickMine. We can use arbitrary contexts, not necessarily tied to the accesses of a particular user [114], known application code paths [34], or certain types of meta-data accesses, which may be client or application specific.

**Black Box Approaches**

Our work is also related to caching/prefetching techniques that treat the storage as a black box, and use fully transparent techniques to infer access patterns [28, 45, 63, 66]. These techniques use sophisticated sequence detection algorithms for detecting application I/O patterns, in spite of access interleaving due to concurrency at the storage server. In this dissertation, we have implemented and compared against two such techniques, *run-based prefetching* [45], and *C-Miner* [61, 62]. We have shown that the high concurrency degree common in e-commerce applications makes these techniques ineffective. We have also argued that QuickMine’s incremental, dynamic approach is the most suitable in modern environments, where the number of applications, and number of clients for each application, hence the degree of concurrency at the storage server vary dynamically.

### 6.3 Dynamic Resource Allocation

The previous work on dynamic resource allocation has primarily focused on the allocation of each resource in isolation: either the CPU, or the memory/cache, or the disk bandwidth among competing workloads. In the following, we discuss the previous work in more detail.

**CPU Scheduling**

The task of properly allocating the CPU has traditionally been the responsibility of the operating system (OS). Within the OS, many algorithms, such as round robin, priority, shortest job first (SJF), lottery scheduling, and multi-level feedback queues have been extensively studied and implemented in variety of operating systems, such as Linux, BSD, and Microsoft Windows [91, 105]. More recently, virtual machines monitors (VMM), e.g., Xen, have implemented their CPU schedulers within the hypervisor to provide performance isolation for their virtual hosts [10].
However, the OS is not aware of the application performance thus it relies on other CPU-centric metrics such as instructions-per-cycle (IPC) to determine the relative benefit of different CPU allocations. The disconnect between the application (i.e., DBMS) and the OS has been studied by commercial database vendors, e.g., Oracle and Microsoft, leading them to implement specialized CPU resource allocators within the DBMS itself [35, 71]. With this functionality, the DBMS allows the database administrator to specify limits CPU limits per application. The administrator typically sets these values using experience, i.e., rules of thumb, or by experimentally trying various settings as the database systems do not automatically control allocation across applications and do not provide allocation across multiple resources.

Memory Partitioning

Dynamic memory partitioning has been studied in the context of operating systems [8, 104, 115] and database systems [13, 14, 24, 69, 100].

Resource containers, implemented within the operating system, provide a mechanism to enforce resource allocation [8]; while this creates a mechanism to account for resource usage, the authors do not provide algorithms to determine the allocations themselves. Waldspurger describes the memory allocation algorithms in the VMWare ESX server [104]. The system administrator sets the bounds of memory usage for each virtual machine. The allocation policy then uses a combination of a share based allocation and an estimate of the working set of each virtual machine. The allocation algorithm also estimates the working-set sizes of each and reclaims the idle memory; with this information, the allocation algorithm periodically adjusts each VM’s allocation to be between the min and max limits. Both of the discussed approaches rely on settings provided by the administrator to determine the memory allocations. However, the most general method to partition memory is to use the miss ratio curve (MRC) [115]. The MRC represents the page miss-ratio versus the memory size, and can be computed dynamically through Mattson’s Stack Algorithm [67]. The algorithm assigns memory increments iteratively to the application with the highest predicted miss-ratio benefit. MRC-based cache partitioning thus dynamically partitions the cache/memory to multiple applications, in such a way to optimize the aggregate miss-ratio.

In the area of database systems, work on memory allocation has focused on two approaches: allocating enough memory to achieve good performance of a single query’s execution and allocating memory to different workloads to ensure performance goals for an application. Mehta and DeWitt show that by controlling when queries are issued to the
DBMS, they can minimize the memory contention between different queries and thus satisfy per-query deadlines [69]. The DBMIN algorithm uses the knowledge of the various patterns of queries to allocate buffer pool memory efficiently [24]. Brown et al. study schemes to ensure per-class response time goals in a system executing queries of multiple classes by sizing the different memory regions [13, 14]. Recently, IBM DB2 added the self-tuning memory manager (STMM) to size different memory regions [100].

However, the above works target only the memory regions within one component, i.e., either the database system, the operating system, or the virtual machine. In this dissertation, we have shown that making local decisions on how to partition caches in a multi-tier environments leads to poor performance. In addition, per-resource performance objectives often run counter to the overall performance goal. We obtain better performance by considering memory resources all tiers of the storage system.

**Disk Bandwidth Partitioning**

Several disk scheduling policies for enforcing disk bandwidth isolation between co-scheduled applications have been proposed and evaluated in previous work [49, 65, 103, 105]. These algorithms primarily use various disk-centric optimization criteria, e.g., time-based deadlines for each I/O [65] or throughput fairness [49].

In more detail, Façade uses a combination of real-time scheduling and feedback-based control of the storage device queue [65]. Façade scheduling is based on Earliest Deadline First (EDF) I/O scheduler and it combines a priority control scheduling with an admission control scheme to dispatch requests of each workload class in such a way that they are serviced according to predefined latency. Jin et al. design the Start-time Fair Queuing (SFQ) algorithm. SFQ uses the notion of virtual time to provide a fair share of the disk bandwidth to all applications sharing the underlying disk [49]. The main drawback of both approaches is that each assumes that the cost of each I/O is known in advance; in general, this knowledge is difficult to ascertain as the performance of the underlying disk varies with disk seeks caused by the workload.

In the course of our work, we have implemented and compared the performance isolation guarantees provided by following disk I/O schedulers: (1) Quanta-based scheduling [103], (2) Start-time Fair Queuing (SFQ) [49], (3) Earliest Deadline First (EDF), (4) Lottery-based [105] and (5) Façade [65]. Our study shows that many of the approaches studied in previous work have two main drawbacks: (i) they rely on disk-centric optimization criteria, and (ii) they constantly switch between workloads causing many unnecessary disk seeks.
leading to poor disk utilization. We find the quanta-based scheduling algorithm proposed by Argon [103] to work best of these algorithms.

Argon partitions the disk bandwidth using a quanta-based scheduler [103]. The quanta-based scheduling algorithm proportionally allocates the disk bandwidth by giving each workload a quantum of time during which it uses the disk exclusively. In particular, our study shows that this offers the strongest isolation between workloads as there are no *disk seeks* between I/Os of two workloads. This is because the quanta-based scheduler allows the storage server to exploit the locality in I/O requests issued by an application during its assigned quantum, which in turn results in minimizing the effects of additional disk seeks due to inter-application interference. We use the quanta-based scheduler in our prototype storage server, *Akash*.

**Multi-resource Partitioning**

Multi-resource partitioning is an emerging area of research where multiple resources are partitioned to provide isolation and QoS for several competing applications. Wachs et al. [103] show the benefit of considering both cache allocation and disk bandwidth allocation to improve the performance in shared storage servers. However, the resource allocation is done after modelling applications through extensive profiling. Chanda et al. [18] implement priority scheduling at the web and database server levels. Wang et al. [107] extend the SFQ [49] algorithm to several storage servers. Padala et al. [75] study methods to allocate memory and CPU to several virtual machines located within the same physical server. However, these works focus on either (i) dynamic partitioning and/or quota enforcement of a single resource on multiple machines [18, 107] or (ii) allocation of multiple resources within a single machine [75, 103]. In this dissertation, we have shown that global resource partitioning of multiple resources located at different tiers results in significant performance gains.

### 6.4 Performance Diagnosis and Validation

Related work in the area of performance diagnosis has focused on three approaches: (1) using statistical correlations, [9, 20, 25, 26, 47, 106], (2) using models [89, 101], and (3) using specialized languages [54, 58, 78, 83].

The statistics based approaches assume that the system is *mostly* correct and detect anomalies as changes from the norm. PeerPressure [106] extends the analysis by comparing
configuration across machines. Pinpoint [20] and Magpie [9] are statistical tools for fault detection in component-based Internet service. Another approach is to use invariants – those metric correlations that hold in a variety of conditions as the correctness measure [47]. Cohen et al. [25, 26] correlate system metrics to high-level states to find the root cause of faults. Unlike our work, these only study simple correlations and statistical deviations, whereas we begin with a high-level hypothesis and analyze how the system’s behavior matches with this hypothesis.

Model-based approaches leverage analytical models provided by the user to contrast system-behavior and localize mismatches [89, 101]. The benefit of this approach is the clear relationship between the metrics and high-level system design. However, developing detailed models is difficult. While our hypotheses require an understanding of the system, we do not require the relationships described by the hypothesis to be always correct, and can inform the user of its validity.

Language based approaches include MACE [54], TLA+ [58], PCL [70], PSpec [78] and Pip [83]. They allow programmers to express their expectations about the system’s communication structure, timing, and resource consumption. PSpec [78] is a performance checking assertion language that allows system designers to verify their expectations about a wide range of performance properties. The type of assertions of PSpec are similar to SelfTalk comparison relations. However, PSpec lacks the ability to use mathematical functions as the basis of checking the behavior of the system. Similar to our work, Pip [83] is an infrastructure for comparing actual behavior and expected behavior of a distributed system, expressed through a declarative language. However, unlike our work, Pip requires the source to be modified by adding some special annotations. In general, in contrast to the existing language based approaches, our work targets system administrators who have a general insight into the system’s behavior but lack the knowledge of the details and have no access to the system’s source code.
Chapter 7

Conclusions and Future Work

In this chapter, we summarize this dissertation by restating the key ideas and techniques. Then, we present some directions on how these techniques can be extended in the future.

7.1 Summary

Modern multi-tier persistent storage systems, the software systems that provide the storage and retrieval of data, are becoming a key component of modern data centers. These systems are designed using commodity components, and are shared among several applications to reduce the operating costs. Balancing the competing imperatives of providing required performance for the applications sharing the infrastructure, on one hand, and maintaining efficient resource usage on the other hand, is a significant research challenge that we address in this dissertation. In order to address this challenge, we design and evaluate methods for predicting, optimizing, and validating the performance of multi-tier persistent storage systems. We show that, given the complexity of modern systems and workloads, the management and performance optimization of these systems by humans becomes increasingly costly and time consuming. To address this acute need, we develop automated techniques for performance tuning, and validation for multi-tier persistent storage systems.

We focus our study on a common persistent storage design, where a database with network-attached storage is shared by several applications. In this setting, we study how to (i) optimize application performance, (ii) dynamically allocate resources to applications, and (iii) validate application performance. To tackle these problems, we introduce two key building blocks: (i) dynamic performance models to estimate resource-to-performance functions quickly and (ii) context-awareness to leverage application and environment features
for performance optimization and dynamic resource allocation.

We first study data prefetching techniques to improve the performance of database applications using network-attached storage. While data prefetching has been used to reduce the storage access latency, the high level of concurrency in today’s applications typically leads to interleaved block accesses, lowering the effectiveness of existing prefetching algorithms. Towards addressing this problem, we propose and evaluate the QuickMine algorithm for context-aware prefetching. The key technique which we develop is context-awareness, where we capture application contexts, such as a transaction or query, and leverage them for context-aware prediction and improved prefetching effectiveness in the storage cache. Our evaluation shows that context-aware prefetching clearly out-performs existing context-oblivious prefetching algorithms, resulting in factors of up to 2 improvement in application latency for two e-commerce workloads with repeatable access patterns, TPC-W and RUBiS.

We next examine techniques for dynamic allocation of resources in shared storage infrastructure. We show that the inherent challenge arises from the interplay between different resources, e.g., changing any cache quota affects the access pattern at the cache/disk levels below it in the storage hierarchy. Our key insight is to incorporate access tracking and known resource dependencies e.g., due to cache replacement policies, into lightweight dynamic performance models. We evaluate our algorithm using both micro-benchmarks, and the industry standard benchmarks: TPC-W and TPC-C. Our results show that multi-resource partitioning allows for performance improvements of up to a factor of 6 for synthetic benchmarks, and a factor of 4 for industry-standard benchmarks compared to state-of-the-art single-resource controllers, and their ad-hoc combination.

Third, we introduce SelfTalk – a declarative language, and Dena – a runtime tool, that in conjunction allow administrators and users to query and validate the performance of multi-tier systems. SelfTalk is sufficiently expressive to encode an administrator’s high level hypotheses/expectations about normal system behavior, such as, “I expect that the throughputs across all system components are linearly correlated”. Given a hypothesis from the administrator, Dena instantiates and validates it using monitored data, gathered within specific system contexts. We evaluate SelfTalk/Dena by posing several hypotheses to validate the performance of a multi-tier storage system, composed of the Akash storage server and the MySQL database server. We find that Dena automatically validates the system performance based on users’ hypotheses and accurately diagnoses system misbehavior; specifically, we validate that the behavior of caches, request schedulers, and the interplay between the different tiers follow the relationships predicted by our dynamic performance models.
7.2 Future Work

In this dissertation, we introduced and leveraged two key building blocks i.e., *context-awareness* and *dynamic performance models* to improve the performance, to dynamically allocate resources, and to validate performance of multi-tier persistent storage systems. We can further leverage these building blocks towards several avenues for future work.

Towards End-to-End Dynamic Resource Allocation

The focus of this dissertation is on multi-tier storage systems running data-intensive applications. This focus allows us to explore the storage issues (e.g., the effect of caching, and interdependency between caching tiers and the disk bandwidth allocation) in depth. However, applications with bottlenecks on other resources exist as well. For example, some applications stress the compute resources i.e., the CPU, others stress the network, etc. Future work in this area is to study how to extend our experience in resource allocation to larger systems, consisting of more tiers.

The immediate future work in this area is to understand the key issues governing the performance of these applications, such as, the effect of scheduling and partitioning of CPU resources. In more detail, we plan to study the problem of end-to-end resource allocation, which involves CPU scheduling, database buffer pool partitioning, and the storage bandwidth partitioning among applications, according to overall performance goals. Specifically, we plan to leverage our experience in the area of multi-tier resource partitioning in the storage hierarchy to build a more comprehensive end-to-end dynamic resource allocation solution [98].

As we expand our focus to much larger systems, the complexity of the system increases dramatically. However, these systems can be modeled in certain operating modes, e.g., a CPU-intensive workload is less affected by the performance of the underlying disk and an I/O intensive workload is less affected by its CPU allocation. Given this insight, we also plan to study if the awareness of different application modes e.g., whether the application is I/O-intensive or CPU-intensive allows us to quickly model and allocate resources. Specifically, we believe that this insight into the operating modes will allow us to model the performance of an application with a much simpler model, i.e., one that takes into account only the performance-critical resource, rather than all resources. Then, we can compose the different simple models to build a *hybrid* model that works for all configurations.
Extending Context-Aware Performance Optimizations

We expect that context-aware caching and prefetching techniques will be of most benefit in modern data center environments, where a fully on-line, incremental technique, robust to changes, and insensitive to the concurrency degree, such as QuickMine, has clear advantages. We believe that our approach can match the needs of many state-of-the-art database and file-based applications. For example, various persistence solutions, such as Berkeley DB or Amazon’s Dynamo [27], use a mapping scheme between logical identifiers and physical block numbers e.g., corresponding to the MD5 hash function [27]. Extending the applicability of our QuickMine algorithm to such logical to physical mappings is a promising area of future work.

Towards Runtime Performance Monitoring

Modern multi-tier storage systems are complex systems leading to complex failure scenarios. In this dissertation, we develop SelfTalk/Dena to diagnose and validate the performance and behavior of multi-tier storage systems; our tools allow the administrator to issue queries, expressing the relationships and behavior of various components to quickly understand and verify the factors affecting the performance of an application. We believe that leveraging the structural and semantic information through the use of hypotheses can be extended to enable flexible runtime monitoring of large systems as well.

An immediate plan for future work is to use the administrator hypothesis as a condition to automatically check system health in the future, and diagnose system anomalies. Specifically, we can explore the concept of hypothesis-based assertions, for detecting system misbehavior, as follows. For a given workload and context, we can save the training information gathered by Dena. Then, at a later time, we can re-evaluate the system using newer monitored data. If the new monitoring data does not match the trained assertion, then we can treat the situation as a potential anomaly in the system and raise an alarm. The key insight is to distinguish benign situations, in which just the parameters of the functional relationship given as hypothesis change, from situations where the relationship itself does not hold. For example, we can first try to re-fit the hypothesis for the current context; if the anomaly is minor, i.e., perturbations in the disk performance or a workload change, then we expect the relationship described in the hypothesis to hold. In this case, no alarm is raised. However, if there is a misbehavior, e.g., a fault in the system, then the re-fit of the hypothesis would fail and we can raise an alarm, stating that the hypothesis failed to match system behavior. A further exploration into these methods of system analysis is an area of future work.
Bibliography


[113] Xifeng Yan, Jiawei Han, and Ramin Afshar. CloSpan: Mining Closed Sequential Patterns in Large Databases. In *Proceedings of the Third SIAM International Conference on Data Mining (SDM’03)*, 2003.