Data-Driven Batch Scheduling

by
John Bent

A dissertation submitted

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

University of Wisconsin - Madison

2005

Abstract

In this thesis, we present a data-driven batch scheduling system. Current CPU-centric batch schedulers ignore the data needs within workloads and execute them by linking them transparently and directly to their needed data. When scheduled on remote computational resources, this elegant solution of direct data access can incur an order of magnitude performance penalty for data-intensive workloads.

To concretely motivate this problem, we provide here a detailed analysis of six current data-intensive, scientific, batch workloads. From this analysis, we derive quantitative bounds on expected scalability and demonstrate the infeasibility of scheduling these workloads using current CPU-centric systems that lack data-awareness.

Adding data-awareness to CPU-centric batch schedulers allows a careful coordination of both data and CPU allocation thereby reducing the performance cost of remote execution. To achieve this coordinated schedule, however, batch schedulers need complicity from storage systems to allow transfer of control over low-level storage decisions from the storage system to the batch scheduler. Wielding explicit storage control, the batch scheduler can then carefully coordinate storage and CPU allocations using a variety of data-driven scheduling policies.

We offer one such system. A modified batch scheduling system that understands the nature of batch workloads as revealed by our new measurement study, that leverages the explicit storage control provided by our new distributed file system, and that can use our new analytical predictive models to select one of the five distinct data-driven scheduling policies that we have created.

Acknowledgments

I would like to begin by thanking my three advisors. Many have expressed their concern to me about the difficulties of having multiple advisors, but I have never regretted it. Each has contributed in a positive fashion in their own way and my experience (and this dissertation) is richer for all three.

I thank Miron for giving a raw anthropology major a chance to work in his batch computing group. Miron has been instrumental in guiding me to the limits of my knowledge and guiding me to the promise of many unknown paths in front of me.

As Miron has shown me the value of hard work, Remzi has shown me its joys. He has helped me select from the unknown paths and has helped me realize that none are as unknown, or challenging, as I might have initially thought. I appreciate Remzi's unique ability to so quickly diagnose misunderstandings; many have been the occasion on which Remzi has found me hopelessly befuddled and left me enlightened.

I thank Andrea for the confidence she has inspired in me. Andrea's standards are rigorous and therefore her approval is empowering. When Miron agreed to work with me, I could disregard it because his funding enables him to work with very many students. When Remzi agreed to work with me, I could spin it to believe that he was willing because we seemed to enjoy each other's company on a personal level. When Andrea agreed to work with me, I finally believed that the raw anthropology major had the potential to write this dissertation.

I'd also like to thank the other members of my committee, Bart Miller and Jon Wolff, for their valuable feedback and suggestions concerning unnoticed related work. Thanks to David DeWitt also for his mentoring and suggestions concerning this work.

I have been exceedingly fortunate further in my colleagues here at Wisconsin both in Miron's Condor group as well as Andrea and Remzi's students. Doug Thain was an early collaborator of mine and was my partner on much of the work contained in this thesis. Doug's cheer and generousity made working many late nights much more palatable.

Muthian Sivathanu, Florentina Popovici, and Nate Burnett have also been outstanding colleagues. I've enjoyed, and will miss, our discussions and our lunches. I wish the best of luck to all of them. Finally, Tim Denehy has been a close friend with whom I've enjoyed many social occasions, technical discussions, and even coding sessions (of which some have even been work related). We also played some sports together but that was much less frequently enjoyable due to large mismatches in ability.

I am grateful for everything that my parents have given me. My father who has always helped me develop my intellectual curiousity and my mother for trying to share with me her special ability to find interesting conversations with anyone. I thank my sister Jenny for being everything an older sibling should be: critical and supportive and loving.

Finally, my immediate family. I love you so very much. I thank my children, Remliik and Eli, for bringing me

so much joy, and pride, and laughter. I am so proud of Remliik for being as wonderful of an older sibling to Eli that Jenny was to me and for understanding when the demands of this dissertation preempted way too many games of catch. I owe you.

Finally, I thank you Charlene, for without you, nothing would be possible.

Contents

Abstract i

Acknowledgments ii

1 Introduction 1
- 1.1 Batch Computing . . . 2
  - 1.1.1 Batch Scheduling Systems . . . 2
  - 1.1.2 Batch Workloads . . . 3
- 1.2 Recent Trends . . . 3
- 1.3 The Problem . . . 4
- 1.4 Our Proposal . . . 6
- 1.5 Our Contributions . . . 6
- 1.6 Organization . . . 8

2 Profiling Batch-Pipeline Workloads 9
- 2.1 Applications . . . 10
- 2.2 Batch-Pipeline Workloads . . . 12
- 2.3 Methodology . . . 14
- 2.4 Workload Analysis . . . 14
  - 2.4.1 Resource Consumption . . . 14
  - 2.4.2 I/O Volumes . . . 15
  - 2.4.3 I/O Operations . . . 16
  - 2.4.4 I/O Types . . . 16
  - 2.4.5 Caching Implications . . . 17
  - 2.4.6 Phase behavior . . . 18
  - 2.4.7 Balanced System Implications . . . 19
- 2.5 System Implications . . . 19
  - 2.5.1 Endpoint Scalability . . . 20
  - 2.5.2 Software Architecture . . . 21
- 2.6 Discussion . . . 25

3 Distributed File Systems for Batch-Pipeline Workloads 26
- 3.1 Architecture . . . 27
  - 3.1.1 Storage Servers . . . 28
  - 3.1.2 Interposition Agents . . . 30
  - 3.1.3 The Scheduler . . . 31
  - 3.1.4 Practical Issues . . . 33
- 3.2 User Burden . . . 33
- 3.3 Experimental Evaluation . . . 34
  - 3.3.1 Methodology . . . 34

3.3.2 I/O Scoping . . . 35
3.3.3 Capacity-Aware Scheduling . . . 36
3.3.4 Batch-intensive Capacity-Aware Scheduling . . . 36
3.3.5 Pipe-intensive Capacity-Aware Scheduling . . . 38
3.3.6 Failure Handling . . . 39
3.3.7 Workload Experience . . . 39
3.3.8 In the Wild . . . 41
3.4 Discussion . . . 43

4 Scheduling Batch-Pipeline Workloads 45
4.1 Methodology . . . 45
4.2 Defining Scheduling Allocations . . . 46
4.2.1 Necessary Scheduling Information . . . 46
4.2.2 Canonical Workloads . . . 47
4.2.3 Compute Environment . . . 47
4.2.4 Derived Values . . . 48
4.2.5 Scheduling Objective . . . 49
4.2.6 Defining Possible Scheduling Allocations . . . 50
4.3 Possible Volume Sizes for the Scheduling Allocations . . . 59
4.4 Predictive Analytical Modelling . . . 61
4.4.1 Predicting Runtime for All . . . 62
4.4.2 Predicting Runtime for AllPrivate . . . 65
4.4.3 Predicting Runtime for AllBatch . . . 67
4.4.4 Predicting Runtime for Slice . . . 68
4.4.5 Predicting Runtime for Minimal . . . 72
4.4.6 Predicting the Effect of Failure . . . 73
4.4.7 Winnowing the Allocations . . . 74
4.4.8 Synthetic Workloads . . . 74
4.4.9 Allocation Behavior . . . 75
4.5 Evaluation . . . 78
4.5.1 Sensitivity to Workload Width . . . 81
4.5.2 Sensitivity to Workload Depth . . . 84
4.5.3 Sensitivity to Batch Volume Size . . . 86
4.5.4 Sensitivity to Private Volume Size . . . 86
4.5.5 Sensitivity to Computation Time . . . 89
4.5.6 Sensitivity to Runtime Variability . . . 89
4.5.7 Sensitivity to Runtime Variability for CPU-intensive Workloads . . . 89
4.5.8 Sensitivity to Network Bandwidths . . . 93
4.5.9 Sensitivity to Failure . . . 93
4.6 Discussion . . . 96

5 Related Work 98
5.1 Profiling . . . 98
5.2 Distributed File-Systems . . . 99
5.3 Scheduling . . . 100
5.3.1 Parallel Scheduling . . . 100
5.3.2 Database Query Planning . . . 101
5.3.3 Data-Driven Batch Scheduling . . . 102

6 Conclusions **103**
6.1 Summary . . . 103
6.1.1 Profiling . . . 103
6.1.2 File System Support . . . 104
6.1.3 Data-Driven Scheduling . . . 105
6.2 Reflections . . . 105
6.2.1 Methodology . . . 106
6.2.2 Research Cycle . . . 106
6.3 Future Work . . . 106
6.3.1 More measurement . . . 107
6.3.2 Non-canonical Workloads . . . 107
6.3.3 Multi-* Effects . . . 108
6.3.4 Inaccurate Information . . . 109
6.3.5 Dynamic Reallocation . . . 110
6.3.6 Checkpointing . . . 111
6.3.7 Partial Results . . . 111
6.4 Postscript . . . 111

Bibliography **113**

Appendices **121**
A.1 Validating the Simulator . . . 121
A.1.1 Validating Functional Correctness of the Workload . . . 121
A.1.2 Validating Functional Correctness of the Compute System . . . 121
A.1.3 Validating Functional Correctness of the Scheduler . . . 122
A.1.4 Validating Performance Accuracy . . . 122
A.1.5 Verifying Against the BAD-FS Implementation . . . 125

List of Tables

2.1 Resources Consumed. 15
2.2 I/O Volume. 16
2.3 I/O Instruction Mix. 17
2.4 I/O Types. 18
2.5 Balanced System Ratios. 24

4.1 Glossary of Canonical Batch-Pipeline Scheduling Terminology. 49
4.2 Minimum Storage Needed. 60
4.3 Effect of W_{Width} and W_{Depth} on Maximum Volume Sizes. 62
4.4 Baseline Characteristics of the Synthetic Workloads. 74
4.5 Baseline Characteristics of the Compute Environment. 76

List of Figures

1.1 Slowdown in Total Time To Completion. . . . 2
1.2 Worldwide Batch Computing Resources. . . . 4
1.3 The Target Environment. . . . 5

2.1 Application Schematics. . . . 11
2.2 A Batch-Pipeline Workload. . . . 13
2.3 Batch Cache Simulation 19
2.4 Pipeline Cache Simulation 20
2.5 I/O Size Distributions 21
2.6 Application Timelines 22
2.7 Scalability of I/O Roles. . . . 23

3.1 System Architecture. . . . 28
3.2 Workflow and Scheduler Examples. . . . 30
3.3 I/O Scoping: Traffic Reduction and Run Times. . . . 35
3.4 Batch-intensive Explicit Storage Management. . . . 37
3.5 Pipe-intensive Explicit Storage Management. . . . 38
3.6 Failure Handling. . . . 39
3.7 Workload Experience. . . . 40
3.8 In the Wild 42

4.1 A Canonical Batch-Pipeline Workload. . . . 48
4.2 The Target Compute Environment. . . . 50
4.3 Minimum Allocated Volumes for each Allocation Strategy 50
4.4 Workflow Traversal for All. . . . 51
4.5 Workflow Traversal for AllPrivate with $V_{Batch} = 1$. . . . 53
4.6 Workflow Traversal for AllBatch with $V_{Exec} = 1$. . . . 55
4.7 Workflow Traversal for Slice with $V_{Exec} = 1$. . . . 56
4.8 Workflow Traversal for Minimal with $V_{Exec} = 1$. . . . 58
4.9 Possible Volume Sizes for the Different Allocations. . . . 61
4.10 The Predictive Model for the All Allocation. . . . 63
4.11 The Predictive Model for the AllPrivate Allocation. . . . 66
4.12 The Predictive Model for the AllBatch Allocation. . . . 67
4.13 The Predictive Model for the Slice Allocation. . . . 69
4.14 The Predictive Model for the Minimal Allocation. . . . 71
4.15 Modelling the Effect of Failure. . . . 73
4.16 Possible Allocations for the Synthetic Workloads. . . . 75
4.17 The CPU and Storage Allocations for the batch-intensive Workload. . . . 77
4.18 The CPU and Storage Allocations for the private-intensive Workload. . . . 79
4.19 The CPU and Storage Allocations for the mixed Workload. . . . 80
4.20 Sensitivity to Workload Width (x-axis is $W_{Width} : C_{CPU}$). . . . 82

4.21 Sensitivity to Workload Depth (x-axis is W_{Depth}). 85
4.22 Sensitivity to Batch Volume Size (x-axis is $W_{Batch} : C_{Storage}$). 87
4.23 Sensitivity to Private Volume Size (x-axis is $W_{Private} : C_{Storage}$). 88
4.24 Sensitivity to Compute Times (x-axis is W_{Run}). 90
4.25 Sensitivity to Runtime Variability (x-axis is $W_{Var} : W_{Run}$). 91
4.26 Sensitivity to Runtime Variability for CPU Intensive Jobs (x-axis is $W_{Var} : (W_{Run} * 1000)$). . . . 92
4.27 Sensitivity to Remote Bandwidth (x-axis is $C_{Remote} : C_{Local}$). 94
4.28 Sensitivity to Failure Rate (x-axis is $C_{Failure}$). 95

A.1 Validating Functional Correctness of the Workload Representation. 122
A.2 Validating Functional Correctness of the Compute System and the Scheduler. 123
A.3 Validating the Performance Correctness of the Simulator. 123
A.4 Checking Simulator Performance against the BAD-FS Implementation. 125

Chapter 1

Introduction

Scheduling batch workloads in a distributed environment is challenging. So challenging, in fact, that the data needs of these workloads are mostly ignored by current batch schedulers. Batch schedulers can ignore these data requirements by transparently transforming distributed environments to resemble the home environments of executing programs. They perform this transformation by using interpostion and indirection. In doing so, they enable a technique called remote I/O in which I/O is redirected such that the programs access remote data directly.

Instead of considering the data requirements, batch schedulers are CPU-centric in that they consider only the computational needs of batch workloads. Data movement in these CPU-centric scheduling systems therefore happens as an unplanned side-effect of job placement. As a job executes and initiates I/O operations, only then does data flow to and from the job.

For many years, this approach has worked well and many important problems in genomics [5], video production [101], simulation [22], document processing [36], data mining [4], electronic design automation [35], financial services [81], and graphics rendering [64] have been solved using the increased computational power offered by batch computing.

However, two recent trends now threaten this technology. First, recent innovations in grid computing allow users, and batch schedulers, access to an increasingly distributed set of remote computational resources. The second trend is that datasets are increasing in size and this growth has been observed to outpace the corresponding increase in the ability of computational systems to transport and process data [53]. Techniques which once worked well for CPU-intensive workloads in a local environment can suffer orders of magnitude losses in throughput when applied to data-intensive workloads in remote environments.

In this dissertation, we offer one approach to this problem: *data-driven batch schedulers* which consider equally the data and CPU requirements of batch workloads.

As Denning noted in early computer systems that memory management and process scheduling must be considered in a coordinated fashion [34], here we make the similar observation that batch scheduling systems can not schedule their workloads independently of storage management.

Data-driven batch schedulers can coordinate the allocation of storage with the allocation of CPU and thereby avoid the large performance drop-offs suffered by current techniques. Developing these new data-driven schedulers is challenging however. As we will see, data-driven batch schedulers require some additional basic workload information, support from the storage system, new scheduling policies, and mechanisms to guide policy selection.

Figure 1.1 graphically illustrates the value of our contribution. Shown are the slowdowns in total time to

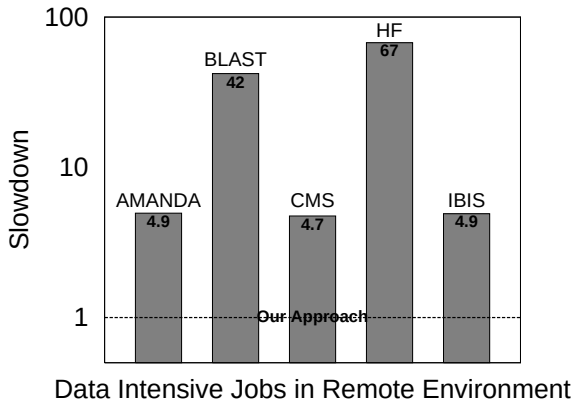


Figure 1.1: **Slowdown in Total Time To Completion.** *This graph depicts the best-case performance for running five data-intensive applications on remote resources. The slowdown shown here is relative to our approach and occurs when these workloads are scheduled using existing CPU-centric batch schedulers which use remote I/O to bind applications to their data. For these experiments, we are showing the slowdown in total time to completion for running 64 instances of each application on a cluster of sixteen compute nodes.*

completion for five scientific batch workloads when the workloads are scheduled using the current CPU-centric scheduling approach as opposed to our new data-driven approach.

In the remainder of this chapter, we will more carefully define what is meant by a batch scheduler, a batch scheduling system, and batch workloads. We will examine how changing trends have affected current batch systems and have created the challenge we are addressing here. We then offer our proposal for data-driven batch schedulers and discuss our contributions towards answering this proposal. We conclude with a brief organizational overview of the rest of the dissertation.

1.1 Batch Computing

1.1.1 Batch Scheduling Systems

Batch computing refers to a system of computing in which a user does not directly dispatch programs interactively for execution but rather delegates this responsibility to a batch scheduling system. The batch scheduler then in turn dispatches the programs for execution, monitors their status, and returns their output upon completion to the user. Although batch schedulers are typically software programs themselves, and we will refer to them as such in this dissertation, this is not necessarily the case; in fact, in the early days of card reading computers, the "batch scheduler" was often a human operator responsible for feeding cards into a computer [99].

Due to this separation between the user and the programs, batch computing is not generally well-suited nor designed for applications which require frequent interactions with the users. More appropriate are long running programs that do not require user input after initialization. Batch schedulers are appreciated by users because they assume the drudgery of program dispatching and monitoring and free the user for more creative endeavors.

In addition to being useful for executing and monitoring *long running* programs, batch schedulers are useful when users have *multiple* programs to execute. Here again, batch schedulers can assume the drudgery and the time-consuming process of dispatching, monitor, and collecting the output of these multiple programs.

Finally, batch schedulers are extremely useful in distributed computational settings where they can dispatch

multiple programs in parallel across multiple computational resources. In such a case, in addition to its other duties of dispatching, monitoring, and collecting output, the batch scheduler is also responsible for monitoring a collection of computational resources and implementing a scheme for matching computational resources with the programs that need them.

1.1.2 Batch Workloads

To use batch scheduling systems, users describe the jobs they want executed along with instructions about how to execute these jobs. These instructions may contain information about external libraries that the programs need, or arguments to be passed to the programs, or information about the input and output paths for job I/O. Some scheduling systems allow the user to supply additional requirements or preferences about the execution environment for their jobs such as preferring machines with at least a gigabyte of memory or requiring machines with file system access to particular files [85].

Although a batch workload might consist of only a single job, it is more typical for many such jobs to be described together and submitted to the batch system within a single workload. There are several reasons why a user might submit several jobs simultaneously instead of just a single job.

One such reason might be that the user has now automated, into a single batch submission, work that was previously done interactively. For instance, this work might consist of running a single job to produce some output data, then running a second job to perform some transformation of the data produced by the first job. For example, the user might be using proprietary software that can not be modified for one of these jobs. Therefore, instead of submitting a single job to the batch system, the user submits a "vertical" sequence of jobs which have an ordering imposed between them. This ordering must be preserved due to the data semantics of the jobs; the child cannot correctly execute until the parent has completely produced the output data which becomes input to the child.

Additionally, users often submit multiple vertical sequences simultaneously such that their workloads consist of a two-dimensional structure of some number of vertical sequences of jobs. Typically, each vertical sequence of jobs in a workload is comprised of the same set of jobs. The difference between each vertical sequence is that each is initiated with some difference in input parameters or input files. For example, users often submit the same vertical sequences of jobs multiple times to perform a parameter sweep [70].

1.2 Recent Trends

Because of the tremendous success that CPU-centric batch schedulers have had, they have attracted a larger, and more diverse, set of users. As with any successful technology, later users are less likely to be technically savvy and are more likely to push the technology beyond its original design. The result is that CPU-centric batch schedulers are being increasingly used to schedule data-intensive workloads submitted by users who may not understand this mismatch.

This mismatch between CPU-centric scheduling and data-intensive workloads is further exacerbated by two emerging trends in distributed computing: an increased utilization of remote resources, and growth in dataset sizes.

Users, and batch scheduling systems, are increasingly likely to have access to remote computational resources. Although many users and organizations have sufficient local computational resources to execute their workloads, there will always exist some set of users that wish to schedule their workloads on remote resources. The growth

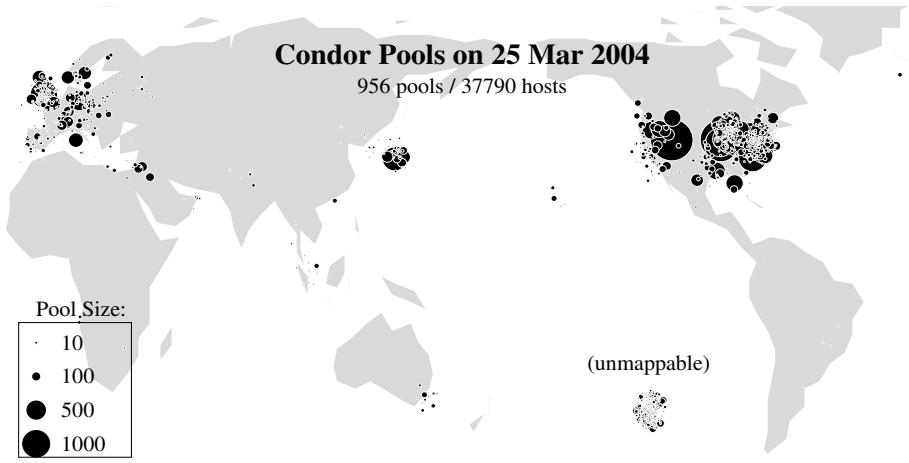


Figure 1.2: **Worldwide Batch Computing Resources.** *This map shows a sampling of the batch computing resources that were available within only the Condor scheduling system in March of 2004. This one snapshot contains almost 40,000 nodes spread over roughly 1,000 clusters. Reproduced with permission from [108].*

and success of grid computing [45] is responsible for this increased access to remote resources and is attested to by the recent and wide-spread media attention surrounding grid computing [28, 71, 72, 105].

An example of the availability of wide-area resource sharing is seen in Figure 1.2. This figure is a snapshot from March 2004 of only a small subset of clusters available for batch computing [108]. This subset contains only clusters running the Condor system which have voluntarily agreed to be included in this measurement. Although the total is difficult to estimate, this subset alone contains almost 40,000 nodes spread over roughly 1,000 clusters.

The second exacerbating trend has been a recent explosive growth in dataset size. The size of users' files has been reported to increase by an order of magnitude three times in the last twenty-five years [13, 89, 118]. Similar observations have been made in batch computing as well [43, 53] with the most cited example being the data to be produced by the Large Hadon Collider at the European physics center CERN [1]. When this operation commences in 2007, it is expected to produce several petabytes of both raw and derived data annually for approximately 15 years.

Difficulties due to the increasing size of data are further exacerbated because this growth has been observed to outpace the corresponding increase in the abilities of computational systems to move and process data [53].

1.3 The Problem

Current CPU-centric batch schedulers are ill-equipped to handle these emerging trends. Ignoring the I/O of workloads and allowing it to occur as an unplanned side-effect of job placement can result in significant throughput loss when scheduling data-intensive workloads on remote resources.

Consider, for example, a user who wishes to run a batch workload in this environment. After developing and debugging the workload on a home system, the user is ready to run batches of hundreds or thousands on all available computing resources, using remote batch execution systems such as Condor [69], LSF [119], PBS [109], or Grid Engine [104]. In Figure 1.3 is a depiction of this situation in which the user's jobs are executing on several machines in different compute clusters scattered across the wide-area network.

Each job in the user's workload is expected to use much of the same input data, while varying parameters

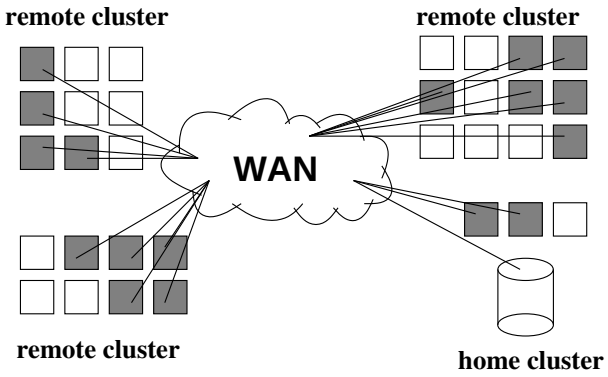


Figure 1.3: **The Target Environment.** *The figure presents a depiction of a typical CPU sharing environment. The user has their important data stored on a storage server in their home cluster, and is running jobs on machines at geographically-dispersed clusters. Each box represents a machine, and shaded boxes are machines upon which the user is currently running jobs. Input data and output data in this environment must be moved between the home storage server and the jobs; temporary data need not be returned to the home storage server.*

and other small inputs. The necessary input data begins on the user's *home storage server* (*e.g.*, an FTP server), and the output data, when generated, should eventually be committed to this home server. Current CPU-centric batch computing systems present a user with two options for running a workload: remote I/O, which is the default behavior, and pre-staging which requires additional, and substantial, user effort.

The first option, remote I/O, is to simply submit the workload to the remote batch system and allow direct processing on remote data. With this option, all input and output occur on demand back to the home storage device. Although this approach is simple, the throughput of a data-intensive workload will be drastically reduced by two factors. First, wide-area network bandwidth is not sufficient to handle simultaneous reads from many data-intensive pipelines running in parallel. Second, all output is directed back to the home site, including temporary data that is not needed after the computation completes.

The second option, pre-staging, is for the user to manually configure the system to replicate input data sets in the remote environment. This approach requires the user to have or obtain an account in the remote environment, identify the necessary input data, transfer that data to the remote site, log into the remote system, unpack the data in an appropriate location, configure the workload to recognize the correct directories (possibly using `/tmp` for temporary data), submit the workload, and manually deal with any failures.

The entire process must be repeated whenever more data needs to be processed, new batch systems become available, or existing systems no longer have capacity to offer to the user. As is obvious from the description, this configuration process is labor-intensive and error-prone; additionally, using `/tmp` can be challenging because its availability cannot be guaranteed. Another limitation is that because the user has made these configurations independently of the scheduling system, the scheduling system is not able to correctly checkpoint job output data. Still, many users go to these lengths simply to run their workloads.

Traditional distributed file systems would be a better solution but are typically not available due to administrative desire to preserve autonomy across domain boundaries. Even were such systems available, their fixed policies prevent them from being viable for data-intensive batch workloads. Consider, for example, BLAST [5], a commonly used genomic search program, consisting of a single job that searches through a large shared dataset for protein string matches.

Assume a user were to run BLAST on a compute cluster of 100 nodes equipped with a conventional distributed

file system such as AFS or NFS. With cold caches, all 100 nodes will individually (and likely simultaneously) access the home server with the same large demands, resulting in poor performance as the dataset is redundantly transferred over the wide area network. Once the caches are loaded, each node will run at local disk speeds, but only if the dataset can fit in its cache. If it cannot, the node will thrash and generate an enormous amount of repetitive traffic back to the home server. Further, lacking workload information, each node must employ some mechanism to protect the consistency and availability of its cached data.

1.4 Our Proposal

To address this problem, we propose *data-driven batch scheduling* in which batch schedulers consider equally the data and CPU requirements of batch workloads. Adding data-driven scheduling to batch schedulers is not straight-forward; we propose that four necessary components to develop these schedulers are currently missing:

- **Workload knowledge.** To develop data-driven batch schedulers requires detail knowledge both about the general structure of batch workloads as well as specific details about the particular workload to be scheduled.
- **Distributed file-system support.** Data-driven batch schedulers cannot implement data-driven policies in isolation. Rather they require support from the distributed file-system such that the batch scheduler, and not the distributed file system, can carefully control exactly how data is allocated.
- **Data-driven scheduling policies.** Of course, having storage control by itself is insufficient. Batch schedulers need new scheduling policies to plan and control the coordinated allocation of data and CPU.
- **Policy selection.** Finally, batch schedulers need some mechanism by which they can make predictions about the relative performance of different data-driven scheduling policies and select the policy that is most appropriate for any particular workload.

1.5 Our Contributions

We offer in this dissertation one such approach to the above proposal. A modified data-driven batch scheduling system that understands the nature of batch workloads as revealed by our new measurement study, that leverages the explicit storage control provided by our new distributed file system, and that can use our new analytical predictive models to select one of the five distinct data-driven scheduling policies that we have created. There are five main contributions in this dissertation:

- **Profiling workloads.** This dissertation is built upon a solid foundation of empirical measurement. In Chapter 2, we present our measurement study of six, important, scientific, data-intensive, batch workloads. In addition to providing a unique and detailed *quantitative* study of batch workloads, these measurements are important because they allow us to *qualitatively* characterize the structure of these workloads and create a taxonomy by which they can be reasoned about.

  We term this structure a *batch-pipeline workload* and use it to characterize the different types of data within batch workloads. We then examine the sharing behavior for each of these types of data. The final contribution of our measurement study is an analysis of the scalability of the six studied applications. We conclude that

batch scheduling systems must understand the batch-pipeline nature of their workloads in order to achieve high degrees of scalability (into the thousands) for data-intensive workloads.

- **Distributed file system support.** Batch scheduling systems cannot efficiently schedule data-intensive workloads in isolation but rather must work in conjunction with a file system that provides transport and storage of data. Traditional distributed file systems, such as NFS and AFS, have been targeted at a particular computing environment, namely a collection of interactively used client machines. However, as past work has demonstrated, different workloads lead to different designs (*e.g.*, FileNet [36] and the Google File System [50]); if assumptions about usage patterns, sharing characteristics, or other aspects of the workload change, one must reexamine the design decisions embedded within distributed file systems.

  In Chapter 4, we make the similar observation that the specific characteristics of batch scheduling systems force a reexamination of the distributed file system. We conclude that current distributed file systems are not well-suited for batch workloads as they hide control of storage decisions such as caching, consistency, and replication. As batch schedulers have access to basic, and semantic, workload information, they are well positioned to make these storage decisions more appropriately. To do so however, batch schedulers need a distributed file system that explicitly allows external control.

  We allow this by introducing a new distributed file system, the Batch-Aware Distributed File System (BAD-FS), and a modified data-driven batch scheduling system. We achieve data-driven batch scheduling by exporting explicit control of storage decisions from the distributed file system to the batch scheduler. Using some simple data-driven scheduling techniques, we demonstrate that our modified system can achieve orders of magnitude throughput improvements both over current distributed file systems such as AFS as well as over current batch scheduling techniques such as remote I/O.

- **Simulation framework.** A minor contribution of our work is the development of a detailed batch scheduling simulation framework, BAD-Sim, for studying data-driven scheduling policies for batch-pipeline workloads. This simulator provides a detailed and nuanced view of the system including the low-level components of the distributed system such as memory, disks, and networks as well as the higher-level services which run upon them. BAD-Sim includes the full machinery of BAD-FS as well as our modified batch scheduling system.

- **Data-driven scheduling.** The fourth major contribution of this dissertation is our study of data-driven policies for scheduling batch-pipeline workloads. Extending on the simple policies we develop to illustrate the importance of external storage control, in Chapter 4 we further develop five distinct, and more complex, data-driven policies each of which leverages external storage control in slightly different ways.

  Multiple policies are needed because there are several performance pitfalls possible when scheduling batch-pipeline workloads in an environment in which storage is insufficient for the complete data needs of the workloads. We identify three such pitfalls. First is an over-utilization of the network connection to refetch data which could otherwise have been cached. Second is an underutilization of the available CPUs due to concurrency limits imposed by data requirements. The third pitfall is an underutilization of the available CPUs due to barriers within the workload.

  Each policy we develop strives to minimize a particular pitfall at the possible expense of the other two. As the relative cost of each pitfall is dependent on different characteristics of the workload and the compute

environment, each policy may be better in some situations and worse in others. The final contribution we make here is to enumerate the set of workload and environmental characteristics which influence the relative performance of the different policies and quantify how each characteristic affects each policy. We discover that what might be the best policy in some situations may in others cause a loss of throughput of over 80%.

- **Predictive modelling.** Having quantified that the different data-driven policies are uniquely sensitive to different workload and environmental characteristics, the final contribution that we make in this dissertation is to create predictive analytical models that estimate the runtime for scheduling a batch-pipeline workload using each particular data-driven policy.

  We then demonstrate the value of these models by comparing them to an ideal model which always makes perfect predictions. Across the range of workload and environmental characteristics for three different synthetic workloads, we find that our model consistently stays within 5% of ideal and never exceeds 30%.

1.6 Organization

The organization of the rest of this dissertation is as follows. In Chapter 2, we provide an in-depth analysis of batch workloads and present a measurement study of six of these workloads from a range of the computational sciences. In Chapter 3, we continue our discussion of why current approaches for scheduling batch-pipeline workloads on remote resources suffer from a range of performance problems and present results illustrating how these problems can be addressed with our new distributed file system BAD-FS. Then in Chapter 4, we examine exactly how the batch scheduler can use this new distributed file system to coordinate the allocation of data with the allocation of compute resources and can use our predictive analytical models to select the appropriate data-driven scheduling policy for a particular workload in a particular compute environment. We compare our dissertation to other related work in Chapter 5 and we conclude and discuss future work in Chapter 6.

Chapter 2

Profiling Batch-Pipeline Workloads

For many years, researchers have studied workload characteristics in order to evaluate their impact on current and future systems architecture [14, 66, 74]. Most of these previous application studies have focused on the detailed behavior of single applications, whether sequential or parallel. For example, the caching behavior of the SPEC workloads has long been a topic of intense scrutiny [22] , and the communication characteristics of parallel applications has similarly been well documented [31, 117, 116].

However, applications are not always used in isolation in production settings. Particularly in computational science, the desired end-result is often the product of a group of applications, each of which may be run hundreds or thousands of times with varied inputs. Such applications are frequently executed in a high throughput batch scheduling system such as Condor [69] and may be managed by high-level workflow software such as Chimera [46].

In this chapter, we present a study of six production scientific workloads. These workloads are from a wide and diverse cross-section of the computational sciences, including astronomy, biology, geology, and physics; further we believe they are representative of an even broader class of important workloads.

We first present a general overview for each of the workloads, describing briefly the scientific goal of each, and presenting their general structures in terms of job and data dependencies. We then define and describe a common structure for these workloads that we term a *batch-pipeline* workload. Batch-pipeline workloads are a well-known abstraction in batch computing, yet they have been heretofore unnamed, only vaguely defined and ill-understood. In this chapter, we create a descriptive taxonomy that allows users, and batch scheduling systems, to strategically reason about the complex dependencies within these workloads.

We then present a thorough characterization of the computational, memory, and I/O demands of these workloads. Although individually a single job from these workloads does not place a tremendous load on system resources, we show that in combination the loads can be overwhelming.

We then characterize the sharing that occurs in the workloads by differentiating between different types of I/O. Through this differentiation, we show that shared I/O is the dominant component of all I/O traffic within these batch-pipeline workloads.

Most importantly, we analyze the implications for systems design and find that wide-area network bandwidth poses a serious scalability problem for these applications, unless attempts are made to eliminate shared I/O. Successful systems for these workloads must segregate the three types of I/O traffic in order to be able to scale successfully. We submit that pipeline data is at least as significant a problem as batch data, and elucidate why traditional

file systems are not appropriate for these workloads. This analysis then leads into Chapter 4 in which we discuss a system we built using these design guidelines.

2.1 Applications

The applications characterized here were chosen from a range of scientific disciplines. The selection criteria were that the applications are attacking a major scientific objective, are composed of sequential applications, and require a scalable computing environment to accomplish high throughput. We focus mostly on six applications but in some measurements SETI@home [102] is included as a point of reference. With guidance from users, we configured the workloads and selected the input parameters to correspond to production use. Diagrams of the applications are shown in Figure 2.1.

BLAST [5] searches genomic databases for matching proteins and nucleotides. Both queries and archived data may include errors or gaps, and acceptable match similarity is parameterized. Exhaustive search is often necessary. A single executable, `blastp`, reads a query sequence, searches through a shared database, and outputs matches.

IBIS [41] is a global-scale simulation of Earth systems. IBIS simulates effects of human activity on the global environment, *i.e.*, global warming. `ibis` performs the simulation and emits a series of snapshots of the global state.

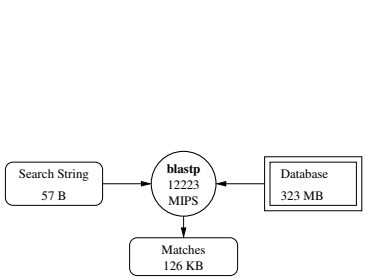
CMS [56] is a high-energy physics experiment to begin operation in 2006. CMS testing software is a two-stage pipeline; the first stage, `cmkin`, given a random seed, generates and models the behavior of accelerated particles. The output is a set of events that are fed to `cmsim`, which simulates the response of the particle detector. The final output represents events that exceed the triggering threshold of the detector.

Nautilus [103] is a simulation of molecular dynamics. An input configuration describes molecules within a three-dimensional space. Newton's equation is solved for each particle. Incremental snapshots are taken to periodically capture particle coordinates. The final snapshot is often passed back to the program as an initial configuration for another simulation. Eventually, all snapshots are converted into a standard format using `bin2coord` and consolidated into images using `rasmol`.

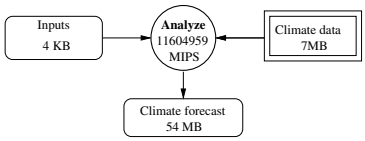
Messkit Hartree-Fock (HF) [30] is a simulation of the non-relativistic interactions between atomic nuclei and electrons, allowing the computation of properties such as bond strengths and reaction energies. Three distinct executables comprise the calculation: `setup` initializes data files from input parameters, `argos` computes and writes integrals corresponding to the atomic configuration, and `scf` iteratively solves the self-consistent field equations.

AMANDA [57] is an astrophysics experiment designed to observe cosmic events such as gamma-ray bursts by collecting the resulting neutrinos through their interaction with the Earth's mass. The first stage of the calibration software, `corsika`, simulates the production of neutrinos and the primary interaction which creates showers of muons. `corama` translates the output into a standard high-energy physics format. `mmc` propagates the muons through the earth and ice while introducing noise from atmospheric sources. Finally, `amasim2` simulates the response of the detector to incident muons.

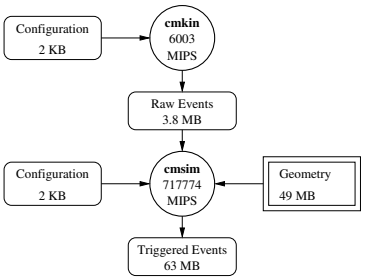
SETI@home is a search for meaningful patterns within space noise which would indicate the presence of extraterrestrial intelligence. Users who wish to participate in this cooperative endeavor allow `SETI` to run as a background process on their computer. This process repeatedly downloads different sections of satellite imagery



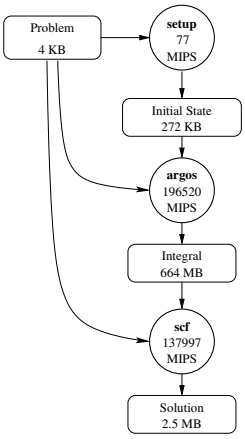
(a) BLAST



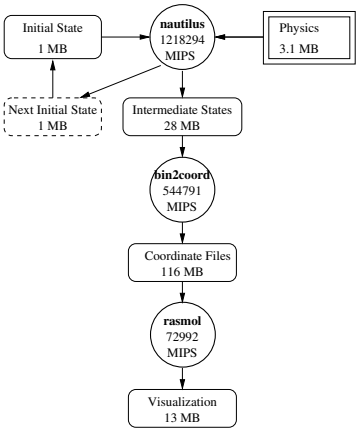
(b) IBIS



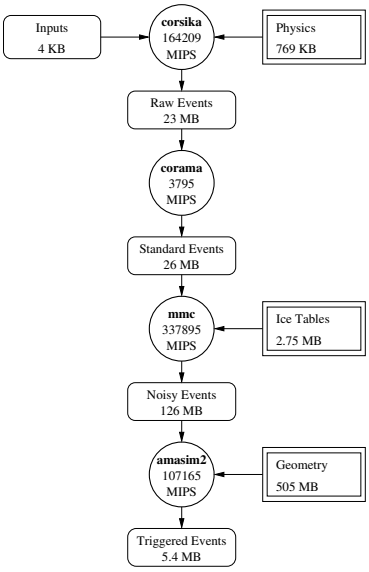
(c) CMS



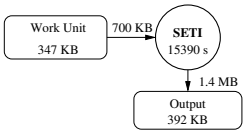
(d) HF



(e) Nautilus



(f) AMANDA



(g) SETI@home

Figure 2.1: **Application Schematics.** *These schematics summarize the structure of each application pipeline. Circles indicate individual processes, labeled with the name and instruction counts. Rounded boxes indicate data private to a pipeline. Double boxes indicate data shared between pipelines in a batch. Arrows indicate data flow.*

and searches within them.

Some of the applications have a variable granularity. Both CMS and AMANDA process a variable number of small, independently generated events. For these applications, we chose pipeline sizes of 250 events (CMS) and 100,000 showers (AMANDA), corresponding to typical production use. In both cases, the CPU and I/O resources consumed by a pipeline scale linearly with the number of events. IBIS has multiple datasets of differing resolutions in which the granularity of the resolution reflects the size of the dataset. In these experiments, we used a medium sized dataset. IBIS and Nautilus perform single simulations of variable length, while SETI, BLAST, and HF operate on a work unit of fixed size.

Across these applications, the following characteristic behaviors were observed:

A diamond-shaped storage profile. Small initial inputs are generally created by humans or initialization tools and expanded by early stages into large intermediate results. These intermediates are often reduced by later stages to small results to be interpreted by humans or incorporated into a database. Intermediate data, which often serves as checkpoint or cached values, may be ephemeral in nature.

Multi-level working sets. Users can easily identify large logical collections of data needed by an application, such as calibration tables and physical constants. However, in a given execution, applications tend to select a small working set of which users are not aware; this has significant consequences for data replication and caching techniques.

Significant data sharing. Although each application has a large configuration space, users submit large numbers of very similar jobs that access similar working sets. For example, analysis of Condor logs shows that the usual batch size is over a thousand for AMANDA, CMS and BLAST. This property can be exploited for efficient wide-area distribution over modest communication links.

2.2 Batch-Pipeline Workloads

One common characteristic of the applications that we studied is that many of them consist of multiple jobs that are logically dependent on each other such that a "child" job can run only after its parent job successfully completes. Within batch computing users often submit a large number of these vertical sequences within a single workload each with slightly different input data or parameters.

We term this two-dimensional structure a *batch-pipeline workload* and present in Figure 2.2 a diagram showing both the structure of this type of workload and the relationships between jobs and data within it. Although not all batch workloads fit this pattern, many do and a study of this type of workload is the focus of this dissertation.

In the diagram, circles represents jobs within the workload and rectangles represent data. There are several things to notice here. First, jobs within a vertical sequence, a *pipeline*, have dependencies between them such that each subsequent job may correctly execute only after the previous job in the pipeline has completed. Notice further how parent jobs "communicate" with their children through the file system by leaving output data which becomes input data to the children.

We classify three distinct types of data within a batch-pipeline workload: pipeline data, endpoint data, and batch data. The first type is the data passed from parent to child which we refer to as *pipeline* data. The second type of data is *endpoint* data which refers to the input data to the initial job in a pipeline and the output data produced by the final job in a pipeline.

For the initial job in a pipeline, the input data is not produced dynamically by a parent during the execution of the workload but rather must be produced prior to the workload's submission. The final job in a pipeline is similar

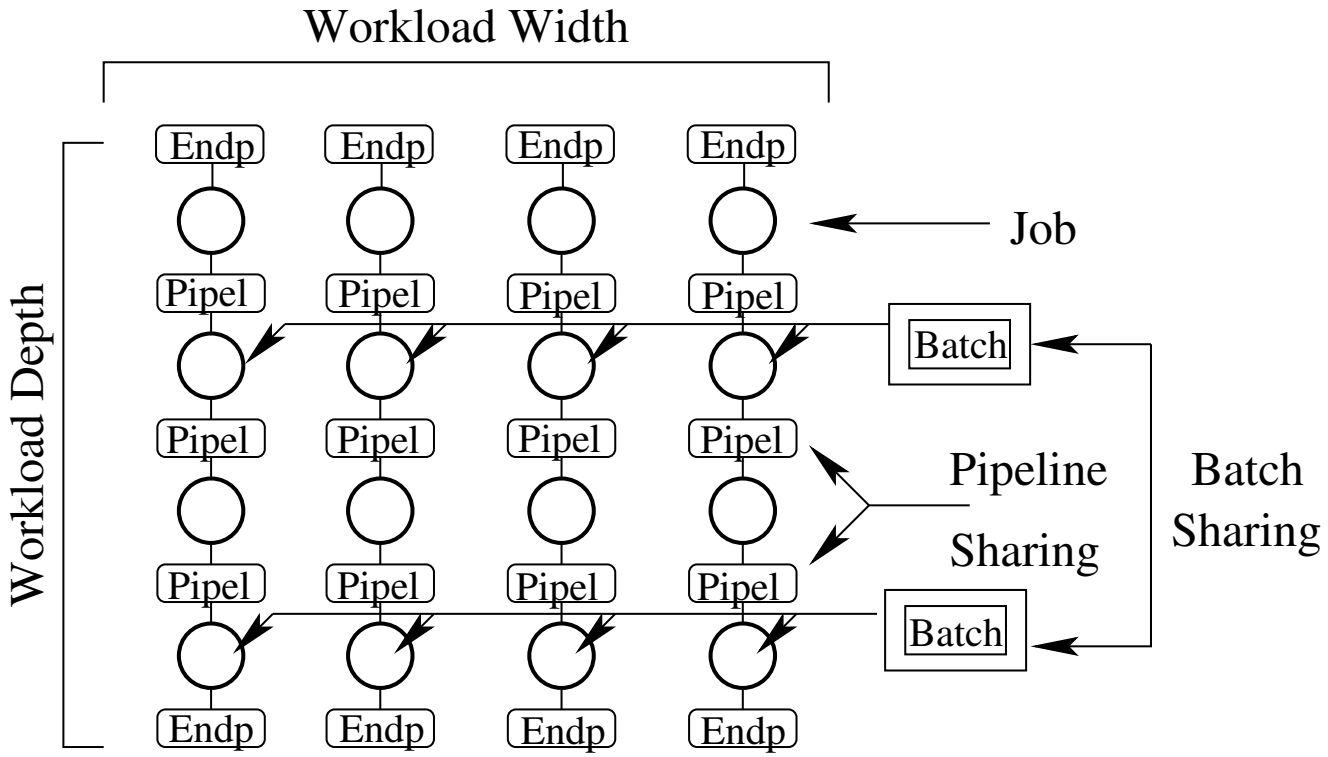


Figure 2.2: **A Batch-Pipeline Workload.** *This diagram shows the two-dimensional structure of a batch-pipeline workload as well as the relationships within it between jobs and data. Circles represent processes and rectangles represent data which are labeled according to data type.*

as it creates endpoint output data which is not consumed by a subsequent child job. The pipeline data passed between jobs is typically not interesting to the user and can be considered temporary data; the endpoint outputs however represent the results of the workload and constitute the output that the batch scheduler returns to the user.

The third type of data is *batch* data which is data read shared by each pipeline. Finally, as a convenience, we sometimes refer in common to both pipeline and endpoint data as *private* data as it is accessed by only a single pipeline as opposed to the batch data which is accessed by multiple pipelines.

By way of example, consider at a high level the BLAST application [5]. BLAST searches for protein and nucleotide string matches within a genomic dataset. The genomic dataset which is searched by each pipeline is batch data. Each pipeline searches for a different protein or nucleotide. Unique command line arguments or input data files instruct each pipeline as to which protein or nucleotide to search for. This unique data is referred to as endpoint input data. The results of the search which contain information as to whether any matches were found and their location within the genome is termed the endpoint output. Finally, any temporary data produced, such as a transformation of the dataset or raw results which were subsequently refined, is the temporary pipeline data.

Finally, we assume that, from the perspective of the batch scheduling system, the jobs within these workloads cannot be modified. In our experience, many scientific workloads are the product of years of fine-tuning, and when complete, are viewed as untouchable. Also, ease of use is important; the less work for the user, the better.

2.3 Methodology

For each application, the CPU, memory and I/O behaviors are captured. The CPU and memory behavior is tracked with available hardware counters and statistics. To instrument I/O behavior, of a shared-library interposition agent [106] is used that replaces the I/O routines in the standard library. For each explicit I/O event requested by the application, the library records an event marking the start and end of the operation, the instruction count, and other details about the I/O request. This technique can be applied to any application that is dynamically linked.

Access to memory-mapped files is traced with a user-level paging technique using the POSIX `mprotect` feature. Access to memory-mapped regions generates a user-level page fault (SIGSEGV) that may be handled and traced by the shared library. Only one application (BLAST) uses memory-mapped I/O. In the analysis that follows, page faults are considered equivalent to explicit read operations of one page size and non-sequential access to memory-mapped pages is recorded as an explicit seek operation.

As with any such technique, the method necessarily interferes with the measurement. This phenomenon, the Heisenberg Effect, is of course not limited to computer science; for example, it is similarly observed in anthropology where it is referred to as the *fishbowl effect* which is evocative of the difficulty of studying a system in which one is also a participant (*i.e.* a fish cannot study the fishbowl).

In this case, the library shares the address space of the target process and contributes to the system resources charged against it. Each traced I/O event generates two I/Os in addition to the actual I/O, so the number of real operations is tripled. The library itself consumes user and system CPU time as it works on behalf of the application, so kernel measurements of these values cannot be relied upon. However, CPU time may be measured independently of the I/O and tracing mechanism by inferring CPU intervals from the gaps between trace events. In the analysis below, “CPU Time” and refers to values computed using these implicit measurements, while “Real Time” refers to the wall clock time while tracing was in effect.

2.4 Workload Analysis

2.4.1 Resource Consumption

An overview of the resources consumed by a single pipeline of each application is given in Table 2.1. Each application may consist of multiple rows in the table; the shading indicates each different application. Subsequent tables are organized in the same way. For example, the BLAST and IBIS applications consist of only a single job and therefore use only one row each in the table whereas CMS, HF, Nautilus, and AMANDA consist of multiple jobs and require multiple rows. The statistics for each application consisting of multiple jobs are then summed in a *total* row.

In this table, real time refers to the total wall-clock time of each of the applications when run without instrumentation overhead. Burst is the average number of instructions executed between I/O operations. We obtained instruction counts using the performance monitoring counters (PMCs) available on x86-class processors. Finally, we measured I/O traffic using an interposition library to intercept and record system calls pertaining to I/O.

There are several things to notice about the resource consumption of these applications. First, these applications have a wide variance in run times on current hardware, ranging from a little more than a minute (BLAST) to a little more than a day (IBIS). Second, considered individually, these applications spend the majority of time consuming CPU rather than I/O. Third, memory requirements and program sizes are all quite modest in comparison to total I/O volume. Finally, notice also that, with the exception of HF, all of the application pipelines have very modest

Application		Real Time (s)	Millions of Instructions			Memory (MB)			I/O Traffic		
			Integer	Float	Burst	Text	Data	Share	MB	Ops	MB/s
seti	seti	41587.1	1953084.8	1523932.2	4.6	0.1	15.7	1.1	75.8	417260	0.00
blast	blastp	264.2	12223.5	0.2	0.1	2.9	323.8	2.0	330.1	88671	1.25
ibis	ibis	88024.3	7215213.8	4389746.8	104.7	0.7	24.0	1.4	336.1	110802	0.00
cms	cmkin	55.4	5260.4	743.8	6.1	19.4	5.0	2.6	7.5	988	0.14
	cmsim	15595.0	492995.8	225679.6	0.4	8.7	70.4	4.3	3798.7	1915559	0.24
	total	15650.4	498256.1	226423.4	0.4	19.4	70.4	4.3	3806.2	1916546	0.24
hf	setup	0.2	76.6	0.4	0.0	0.5	4.0	1.3	9.1	2953	56.43
	argos	597.6	179766.5	26760.7	0.8	0.9	2.5	1.4	663.8	254713	1.11
	scf	19.8	132670.1	5327.6	0.2	0.5	10.3	1.3	3983.4	765562	201.06
	total	617.6	312513.2	32088.6	0.3	0.9	10.3	1.4	4656.3	1023228	7.54
nautilus	nautilus	14047.6	767099.3	451195.0	18.6	0.3	146.6	1.2	270.6	65523	0.02
	bin2coord	395.9	263954.4	280837.2	4.2	0.0	2.2	1.4	403.3	129727	1.02
	rasmol	158.6	69612.8	3380.0	1.9	0.4	4.9	1.7	128.7	38431	0.81
	total	14602.2	1100666.5	735412.2	7.9	0.4	146.6	1.7	802.7	233681	0.05
amanda	corsika	2187.5	160066.5	4203.6	26.4	2.4	6.8	1.4	24.0	6225	0.01
	corama	41.9	3758.4	37.9	0.3	0.5	3.2	1.1	49.4	12693	1.18
	mmc	954.8	330189.1	7706.5	0.3	0.4	22.0	4.9	154.4	1141633	0.16
	amasim2	3601.7	84783.8	20382.7	143.7	22.0	256.6	1.6	550.3	733	0.15
	total	6785.9	578797.8	32330.7	0.5	22.0	256.6	4.9	778.0	1161275	0.11

Table 2.1: **Resources Consumed.** *Shown in Table 2.1 are the total amounts of resources consumed. In this and in subsequent tables, shading is used to differentiate between different application pipelines. Real time refers to the total wall-clock time of each of the applications when run without instrumentation overhead. Burst is the average number of instructions executed between I/O operations. Instruction counts were obtained using the performance monitoring counters (PMCs) available on x86-class processors. Traffic was measured using an interposition library to intercept and record system calls pertaining to I/O.*

bandwidth requirements.

2.4.2 I/O Volumes

Table 2.2 details the I/O volume produced by each pipeline stage. Traffic is the number of bytes that flow into and out of the process. Unique I/O considers only unique byte ranges within this total traffic. Static I/O refers to the total size of all of the files accessed and may be greater than unique I/O if applications read only portions of the files.

Although these applications are conceived as a pipeline of multiple stages, they are not connected by simple data streams. Rather, each makes complex read/write use of the file system, as indicated by the number of files each accesses. Notice here again the wide discrepency between applications which ranges from HF which accesses only 9 unique files to Nautilus which accesses over 250.

Another observations is that SETI, CMS, HF, and to a lesser degree, BLAST, all read input data multiples times as indicated by the discrepancy between the Traffic and Unique rows in the Reads category. This suggests that caching may be particularly important to them.

Further, over-writing of output data is also found in all pipelines with the exception of AMANDA. Output over-writing is usually done to update application-level checkpoints in place. (We are somewhat alarmed to observe that such checkpoints are unsafely written directly over existing data, rather than written to a new file and atomically replaced by renaming it.)

Finally, several pipelines are distributed with large collections of data that may be of use to many runs. However, any typical run only accesses a small portion common to similar runs. For example, the static size of the BLAST dataset exceeds the unique amount read by the application by 45%. This suggests that systems which prestage entire data sets may sometimes be performing unnecessary work.

	Total I/O				Reads				Writes			
Application	**Files**	**Traffic**	**Unique**	**Static**	**Files**	**Traffic**	**Unique**	**Static**	**Files**	**Traffic**	**Unique**	**Static**
seti	14	75.77	3.02	3.02	12	71.62	0.72	1.04	11	4.15	2.36	2.68
blastp	11	330.11	323.59	586.21	10	329.99	323.46	586.09	1	0.12	0.12	0.12
ibis	136	336.08	73.64	73.64	132	140.08	73.48	73.48	118	196.00	66.66	66.66
cmkin	4	7.49	3.88	3.88	2	0.00	0.00	0.00	2	7.49	3.88	3.88
cmsim	16	3798.74	116.00	126.18	11	3735.24	52.86	63.05	5	63.50	63.13	63.13
total	17	3806.22	119.88	130.06	11	3735.24	52.86	63.05	6	70.98	67.01	67.01
setup	5	9.13	0.40	0.40	3	5.44	0.26	0.26	3	3.69	0.39	0.40
argos	5	663.76	663.75	663.97	2	0.04	0.03	0.26	4	663.73	663.74	663.97
scf	11	3983.40	664.61	664.61	9	3979.33	663.79	664.60	8	4.07	2.50	2.69
total	11	4656.30	666.54	666.54	9	3984.81	663.80	664.60	9	671.49	666.53	666.53
nautilus	17	270.64	32.90	32.90	7	4.25	4.25	4.25	10	266.40	28.66	28.66
bin2coord	247	403.27	273.87	273.87	123	152.78	152.66	152.66	241	250.49	249.39	249.39
rasmol	242	128.75	128.76	128.76	124	115.87	115.88	115.88	120	12.88	12.88	12.88
total	501	802.66	435.48	435.48	252	272.90	272.74	272.74	369	529.76	290.94	290.94
corsika	8	23.96	23.96	23.96	5	0.76	0.75	0.75	3	23.21	23.21	23.21
corama	6	49.37	49.37	49.37	3	23.17	23.17	23.17	3	26.20	26.20	26.20
mmc	11	154.36	154.36	154.36	9	28.92	28.92	28.92	2	125.43	125.43	125.43
amasim2	29	550.35	550.40	635.78	27	545.04	545.09	630.47	3	5.31	5.31	5.31
total	46	778.04	778.09	863.42	40	597.89	597.96	683.32	7	180.14	180.11	180.11

Table 2.2: **I/O Volume.** *This table shows the total amounts of I/O performed. Traffic is the number of bytes that flow into and out of the process. Unique I/O considers only unique byte ranges within this total traffic. Static I/O refers to the total size of all of the files accessed and may be greater than unique I/O if applications read only portions of the files.*

2.4.3 I/O Operations

The distribution of I/O operations is given in Table 2.3. The `Seek` column includes non-sequential access to memory-mapped pages and ignores all `lseek` operations which do not actually change the file offset. The `Other` column sums a number of generally uncommon operations such as `ioctl` and `access`.

Notice that many of these applications have a high degree of random access, as shown by the ratio of `seeks` to `reads` and `writes`. This results from the nature of the data files accessed by the programs, generally with complex, self-referencing, internal structure, and contradicts many previous file system studies which indicate the dominance of sequential I/O [13].

This suggests that these applications are not entirely designed with computational performance as their paramount goal. Rather they are presumably designed by scientists in their respective fields who choose to focus on their own science and wish to use computers with as little fuss as possible.

Further these applications are undoubtedly developed incrementally and are presumably initially run a single pipeline at a time, by a single scientist, on a single computer. As the application evolve however it becomes easier to understand and interpret its results and it can be packaged in large submissions for batch computing. Moving now into a distributed environment, the penalty for random access may become significantly higher, especially if it is compounded by concurrent access to the same file by multiple instances of the application.

Finally, the high numbers in the `Other` column reflect the fact that `bin2coord` and `rasmol` are driven by shell scripts which perform many `readdir` operations.

2.4.4 I/O Types

To characterize the different types of sharing in batch-pipelined workloads, we have divided the I/O traffic into three roles as shown earlier in Figure 2.2. *Endpoint* traffic consists of the initial inputs and final outputs that are unique to each pipeline. They must be read from and written to the home storage server regardless of the system design. *Pipeline* traffic consists of intermediate data passed between pipeline stages or even intermediate data

Appl.	Open	(%)	Dup	(%)	Close	(%)	Read	(%)	Write	(%)	Seek	(%)	Stat	(%)	Other	(%)
seti	64595	*16*	0	*0*	64596	*16*	64266	*15*	32872	*8*	63154	*15*	127K	*31*	15	*0*
blastp	18	*0*	11	*0*	18	*0*	84547	*95*	1556	*2*	2478	*3*	37	*0*	5	*0*
ibis	1044	*1*	0	*0*	1044	*1*	26866	*24*	28985	*26*	51527	*47*	1208	*1*	122	*0*
cmkin	2	*0*	0	*0*	2	*0*	2	*0*	492	*50*	479	*49*	8	*1*	2	*0*
cmsim	17	*0*	0	*0*	16	*0*	952859	*48*	18468	*1*	944125	*49*	47	*0*	24	*0*
total	19	*0*	0	*0*	18	*0*	952861	*48*	18960	*1*	944604	*49*	55	*0*	26	*0*
setup	6	*0*	0	*0*	6	*0*	1061	*36*	735	*25*	1118	*38*	19	*1*	6	*0*
argos	3	*0*	0	*0*	3	*0*	8	*0*	127569	*50*	127106	*50*	18	*0*	4	*0*
scf	34	*0*	0	*0*	34	*0*	509642	*67*	922	*0*	254781	*33*	121	*0*	18	*0*
total	43	*0*	0	*0*	43	*0*	510711	*50*	129226	*13*	383005	*37*	158	*0*	28	*0*
nautilus	497	*1*	0	*0*	488	*1*	1095	*2*	62573	*96*	188	*0*	678	*1*	1	*0*
bin2coord	1190	*1*	6977	*5*	12238	*9*	33623	*26*	65109	*50*	3	*0*	407	*0*	10K	*8*
rasmol	359	*1*	22	*0*	517	*1*	29956	*78*	3457	*9*	1	*0*	252	*1*	3K	*10*
total	2046	*1*	6999	*3*	13243	*6*	64674	*28*	131139	*56*	192	*0*	1337	*1*	14K	*6*
corsika	13	*0*	0	*0*	13	*0*	199	*3*	5943	*96*	8	*0*	36	*1*	10	*0*
corama	4	*0*	0	*0*	4	*0*	5936	*47*	6728	*53*	2	*0*	12	*0*	4	*0*
mmc	8	*0*	0	*0*	9	*0*	29906	*3*	1111686	*97*	0	*0*	7	*0*	7	*0*
amasim2	30	*4*	0	*0*	28	*4*	577	*79*	24	*3*	4	*1*	57	*8*	10	*1*
total	55	*0*	0	*0*	54	*0*	36618	*3*	1124381	*97*	14	*0*	112	*0*	31	*0*

Table 2.3: **I/O Instruction Mix.** *Shown here are the total number of the different types of I/O instructions executed by each of the applications. The* `Seek` *column includes non-sequential access to memory-mapped pages and ignores all* `lseek` *operations which do not actually change the file offset. The* `Other` *column sums a number of generally uncommon operations such as* `ioctl` *and* `access`*.*

passed between different phases of a single stage. *Batch* traffic is input data that are identical across all pipelines. Through our understanding of each application, we identified every file accessed as either endpoint, pipeline, or batch, and computed the traffic performed in each category, as shown in Table 2.4.

We immediately see that comparatively little traffic is needed at the endpoints; the bulk is either pipeline or batch, depending on the application. Only IBIS has a significant amount of endpoint traffic relative to its total. This indicates that the scalability of systems that run these applications will depend on their ability to differentiate between these different types of I/O.

Finally, examining across both Tables 2.3 and 2.4, we note that a very large number of `opens` are issued relative to the number of files actually accessed. Typically designed on standalone workstations, these applications are not optimized for the realities of distributed computing, where opening a file for access can be many times more expensive than issuing a read or write.

2.4.5 Caching Implications

To examine the sharing potential of each workload, we separated the pipeline and batch I/O traces and replayed them over cache simulators of various sizes. These traces were obtained by using an interposition library and logging each system call performed. We assume a 4 KB block size in the simulator. Note that the simulator used here is relatively simple and is different from the simulator we discuss in more detail in Chapter 4.

These results are shown in Figures 2.3 and 2.4. Note that we have included the size of the executable files implicitly in these calculations as batch data.

In general, for both types of sharing, the necessary cache sizes are small with respect to both the I/O volume and the sizes of typical main memories today. There are some outliers. AMANDA has a large amount of batch shared data (over half a GB) that is read only once, and thus a cache is not effective until very large sizes. However, AMANDA also has a very high pipeline hit rate at small cache sizes due to a large number of single-byte I/O requests. Due to its high degree of re-reading and output overwriting, CMS needs only very small cache sizes to

	Endpoint I/O (MB)				Pipeline I/O (MB)				Batch I/O (MB)			
Appl.	**Files**	**Traffic**	**Unique**	**Static**	**Files**	**Traffic**	**Unique**	**Static**	**Files**	**Traffic**	**Unique**	**Static**
seti	2	0.34	0.34	0.34	12	75.43	2.68	2.68	0	0.00	0.00	0.00
blastp	2	0.12	0.12	0.12	0	0.00	0.00	0.00	9	329.99	323.46	586.09
ibis	20	179.92	53.97	53.97	99	148.27	12.69	12.69	17	7.89	6.98	6.98
cmkin	2	0.07	0.07	0.07	1	7.42	3.81	3.81	1	0.00	0.00	0.00
cmsim	6	63.50	63.13	63.13	1	5.56	3.81	3.81	9	3729.67	49.04	59.24
total	6	63.56	63.20	63.20	2	12.99	7.62	7.62	9	3729.67	49.04	59.24
setup	3	0.14	0.14	0.14	2	8.99	0.26	0.26	0	0.00	0.00	0.00
argos	3	1.81	1.81	1.81	2	661.95	661.93	662.17	0	0.00	0.00	0.00
scf	3	0.01	0.01	0.01	7	3983.39	664.59	664.59	1	0.00	0.00	0.00
total	3	1.96	1.94	1.94	7	4654.34	664.59	664.59	1	0.00	0.00	0.00
nautilus	6	1.18	1.10	1.10	9	266.32	28.66	28.66	2	3.14	3.14	3.14
bin2coord	1	0.00	0.00	0.00	241	403.25	273.85	273.85	5	0.02	0.01	0.01
rasmol	119	12.88	12.88	12.88	120	115.79	115.79	115.79	3	0.08	0.09	0.09
total	124	14.06	13.99	13.99	369	785.37	418.25	418.25	8	3.24	3.24	3.24
corsika	2	0.04	0.04	0.04	3	23.17	23.17	23.17	3	0.75	0.75	0.75
corama	3	0.00	0.00	0.00	3	49.37	49.37	49.37	0	0.00	0.00	0.00
mmc	0	0.00	0.00	0.00	6	151.63	151.63	151.63	5	2.73	2.73	2.73
amasim2	5	5.31	5.31	5.31	2	40.00	40.00	125.43	22	505.04	505.04	505.04
total	6	5.22	5.21	5.21	11	264.31	264.29	349.69	29	508.52	508.52	508.52

Table 2.4: **I/O Types.** *Shown here are the total amounts of each type of I/O performed. Endpoint traffic consists of the initial inputs and final outputs that are unique to each application. Pipeline traffic is intermediate data passed between pipeline stages or even intermediate data passed between different phases of a single stage. Batch traffic is input data that are shared across different instances of the pipeline. Traffic is the number of bytes that flow into and out of the process. Unique I/O considers only unique byte ranges within this total traffic. Static I/O refers to the total size of all of the files accessed and may be less than unique I/O if applications read only portions of the files.*

effectively maximize its hit rates. BLAST has no pipeline data. IBIS, though one stage, has pipeline data in the form of checkpoints written and read multiple times.

The high hit rates observed here, in some cases approaching 100% are due to two reasons. First, as seen in Table 2.2, many of the applications tend to re-access data frequently. Second, the granularity of the I/O's performed is often smaller than the simulated cache size. In this case, only the first access to a portion of the block registers as a cache miss, subsequent accesses, even to different portions of the block, will then be recorded as cache hits (so long as the block has not been evicted in the interim).

Rational for the high hit rates is evidenced further by examining Figure 2.5 which shows the cumulative distributions of I/O operations performed by each application as a function of the I/O size. Notice here for example that Nautilus and AMANDA in particular perform almost all of their I/O at a granularity smaller than 4 KB.

2.4.6 Phase behavior

Timelines of CPU and I/O activity are shown in Figure 2.6. These timelines show the phase behavior of these applications at both the process level and the pipeline level. For example, BLAST consists of a scanning phase that reads the entire protein database sequentially, and then returns to selected portions non-sequentially, resulting in much lower input bandwidth.

Both BLAST and HF are input-bound for significant intervals, both while performing sequential I/O. IBIS and AMANDA show periodic output bursts, suggesting good opportunities for overlapping CPU and I/O bursts. While not obviously periodic, Nautilus alternates long CPU bursts of several hundred seconds with periods of shorter CPU and output bursts. Clear division in the I/O role at the process level can be seen in HF, Nautilus, and AMANDA.

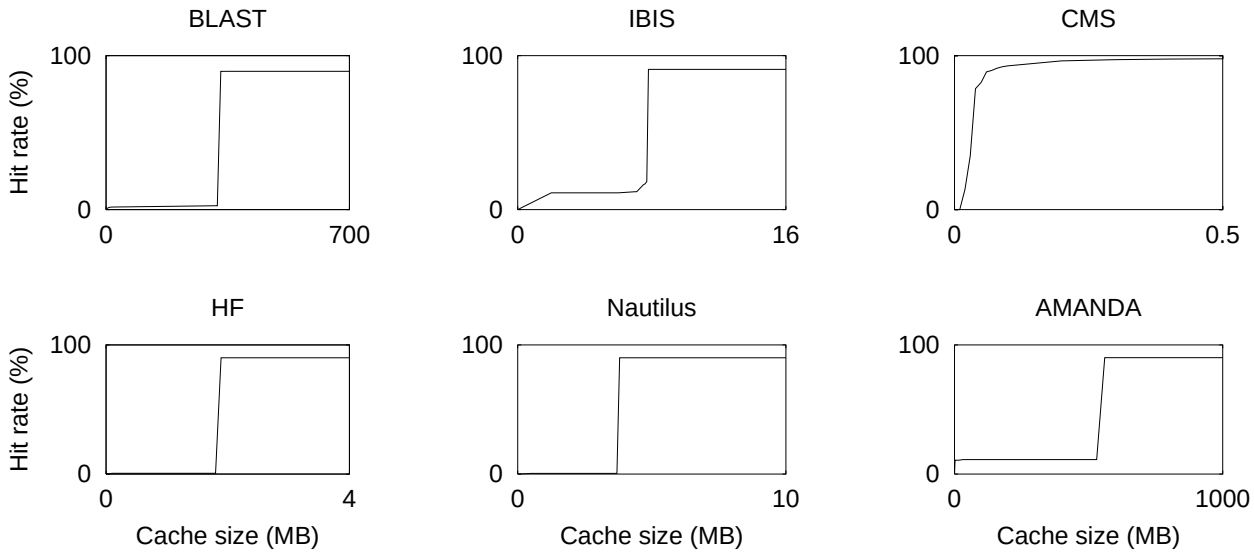


Figure 2.3: **Batch Cache Simulation** *These graphs show the cache hit rates obtained by re-running an I/O trace of the batch data accesses for each application on a simulated cache with a block size of 4 KB.*

2.4.7 Balanced System Implications

Finally we analyze and present in Table 2.5 how these applications relate to Amdahl's long standing system balance ratios [6], recently amended by Gray [52]. These rules of thumb guide the selection of CPU, memory, and I/O resources in the design of a "balanced" system. These workloads have CPU-IO ratios (measured in MIPS/MBPS) far exceeding Amdahl's ideal value of eight, indicating reliance on computation rather than I/O. The ratio of memory to CPU speed, known as alpha, is near or below Amdahl's value of one, and with the exception of the last component of AMANDA, never comes close to Gray's value of four. This also indicates reliance on computation rather than memory. Finally, the ratio of CPU instructions to I/O instructions is several orders of magnitude larger than 50,000. With respect to a single instance of each pipeline, a commodity computing node engineered to Amdahl's metrics is considerably overprovisioned with I/O bandwidth and memory capacity. The aggregation of multiple pipelines however has more significant needs, described below.

2.5 System Implications

Each of these workloads are potentially infinite. In these problem domains, the ability to harness more computing power enables higher resolution, more parameters, and lower statistical uncertainties. Current users of these applications wish to scale up throughput by running hundreds or thousands simultaneously. At this scale, applications normally considered CPU-bound become I/O bound when considered in aggregate.

To give some idea of the growing envelope of current scientific computing, consider that in the spring of 2002, the CMS pipeline was used to simulate 5 million events divided into 20,000 pipelined jobs, consuming 6 CPU-years and producing a terabyte of output. This batch was only a small fraction attempted as a test run before full production begins in 2007. Successive yearly workloads are expected to continuing growing. All the necessary code and data are published in authoritative form by the experiment's central site. Likewise, all simulation outputs must eventually be moved back for archival storage. When the Large Hadron Collider (LHC) [1] at the European

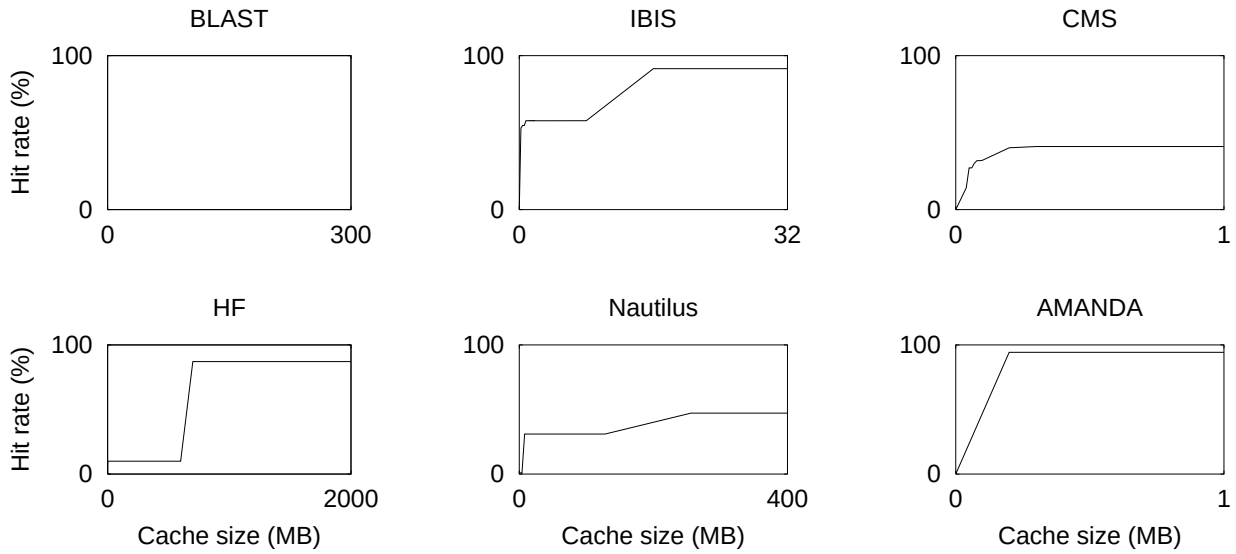


Figure 2.4: **Pipeline Cache Simulation** *These graphs show the cache hit rates obtained by re-running an I/O trace of the pipeline data accesses for each application on a simulated cache with a block size of 4 KB.*

physics center CERN commences this operation in 2007, it is expected to produce several petabytes of both raw *and* derived data annually for approximately 15 years.

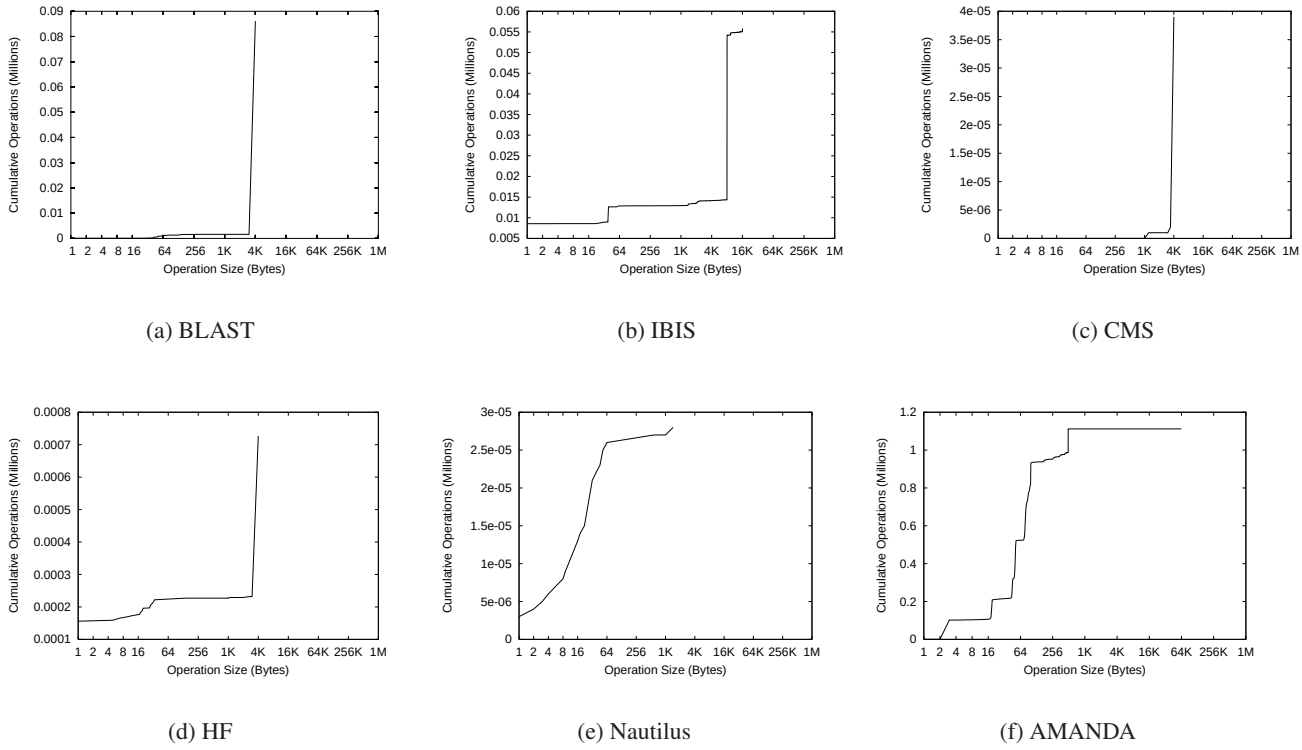
In this section, we will explore the general properties of computing and storage systems that may be built to satisfy these workloads. We consider here the provisioning of the necessary resources and postpone our discussion of designing storage systems for these workloads until Chapter 3 and our discussion of detailed scheduling algorithms for data management until Chapter 4.

2.5.1 Endpoint Scalability

Regardless of the capacity of individual computing nodes, the ultimate scalability of these workloads is limited by competition for shared resources. We assume that each workload relies on a central site for the authenticating and archiving input and output data. However, we have demonstrated that actual endpoint I/O traffic is a relatively small fraction of the total for all of these applications. If we are able to eliminate all non-endpoint traffic from the endpoint server through techniques such as caching and replication then we may see significant gains in scalability.

Of course, traffic elimination must be carried out carefully. Pipeline-shared traffic may only be eliminated if it is truly of no use to the end user. Such intermediate data might be necessary to return for debugging or even for archival if the ability to reproduce it is questionable. Batch-shared may only be eliminated within the constraints of maintaining the consistency and authenticity of potentially changing input data. Traffic elimination cannot be done blindly without some consideration of how the data are actually used outside the computing system.

That said, we may consider the limits of a system for executing such workloads based on its ability to eliminate shared traffic. Figure 2.7 shows how each of the selected applications would scale in four systems each eliminating some category of traffic. We assume the presence of a buffering structure sufficient to completely overlap all CPU and I/O; figures assume a 2000 MIPS CPU and show MB per second of CPU time. Four horizontal lines show comtemporary milestones (as of 2005) in I/O bandwidth. The lowest, at 1 MB/s, represents wide-area network bandwidth, the next, at 12 MB/s, represents local-area network bandwidth, the next, at 40 MB/s, represents a



(a) BLAST (b) IBIS (c) CMS

(d) HF (e) Nautilus (f) AMANDA

Figure 2.5: **I/O Size Distributions** *These graphs show the cumulative distribution of I/O operations as a function of I/O size. For example, almost all of the 400,000 operations performed by CMS are at the 4 KBsize. Conversely, Nautilus and AMANDA performed almost no operations at this size or larger.*

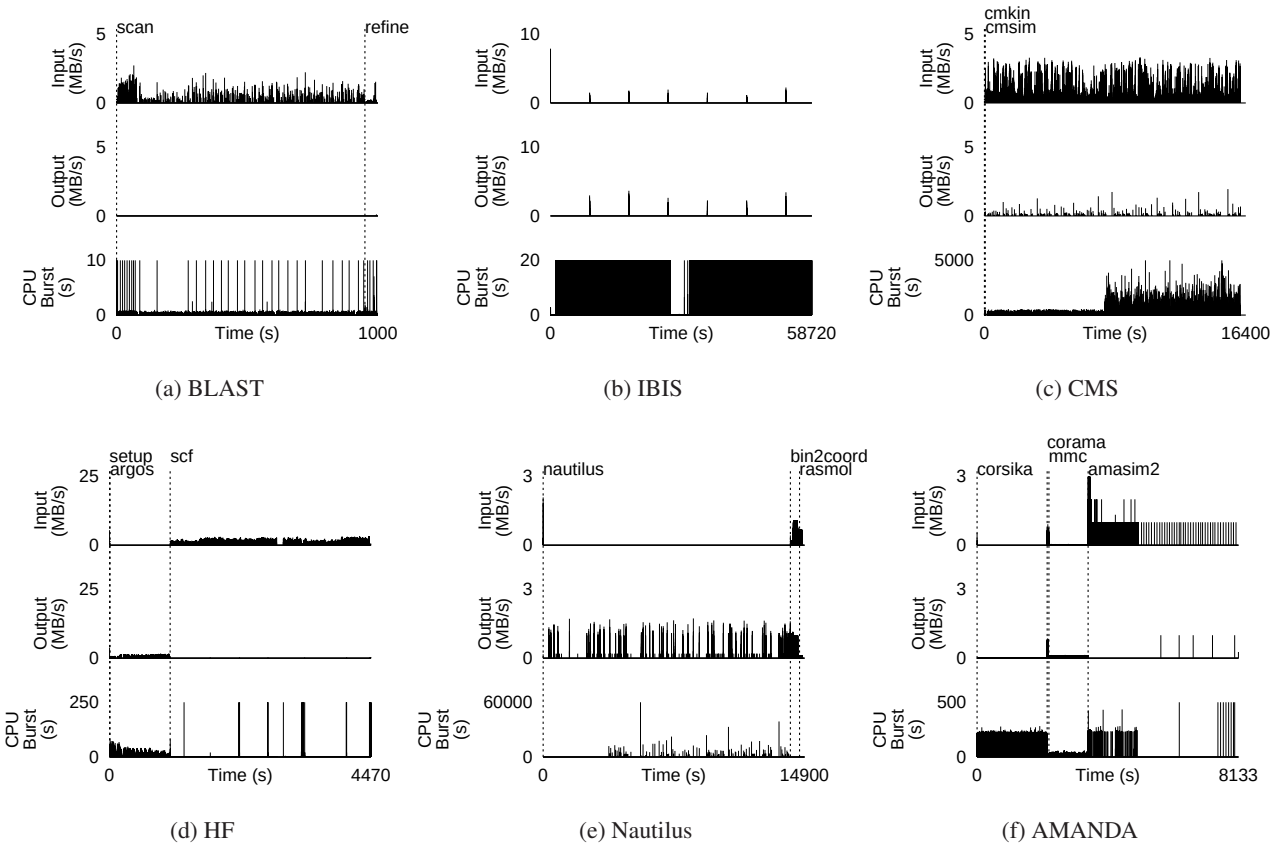
commodity hard disk, and finally, the uppermost, at 1500 MB/s, represents a a high-end storage server and network.

The leftmost graph shows the scalability of a system that carries all traffic to the endpoint server. In this discipline, a high end storage device is needed for systems of very modest size. Only IBIS and SETI would be able to scale to a width approaching 100,000. If batch-shared traffic is eliminated, we will make significant improvements in CMS and Nautilus, as shown in the second graph. On the other hand, if pipeline-shared traffic is eliminated, we observe significant gains for SETI, HF, and Nautilus, as shown in the third. Once both batch and pipeline data is eliminated and only endpoint I/O is considered, then we reach the limit shown in the final graph. All of the applications shown could scale to a width greater than 100 across the wide-area network, 1000 with a commodity storage server in a local-area network, and over 100,000 with high-end storage. SETI alone could potentially scale to 1 million CPUs, an indicator of its specialized design for wide-area deployment.

2.5.2 Software Architecture

In order to scale to large sizes, software architectures for these workloads must strive to eliminate batch-shared and pipeline-shared data from endpoint interactions wherever possible, within the constraints of security, persistence, and performance. Traditional file systems do not serve these applications because their naming and consistency requirements are targeted to interactive cooperating users. These applications require a data management system that has specialized requirements for workload analysis, failure recovery, and resource management.

By itself, the issue of batch input sharing has received significant attention in the grid computing community.



(a) BLAST (b) IBIS (c) CMS

(d) HF (e) Nautilus (f) AMANDA

Figure 2.6: **Application Timelines** *These figures show the I/O and CPU utilization of the applications as a function of time and are designed to reveal any phase behavior either between jobs within the application or within individual jobs themselves.*

Deployed systems such as SRB [83] and GDMP [94] manage widely-distributed and well-known batch shared data. Stork [60] is a more recently proposed system which introduces powerful data transfer techniques which the user can employ for both batch data and for endpoint data. Additional techniques for discovering [112] and replicating [86] batch data have been proposed as well.

Without diminishing the importance of batch sharing, the issue of pipeline sharing is a very different problem that has been relatively neglected. As Figure 2.7 shows, the localization of both types of I/O is necessary to achieve high scalability. The treatment of pipeline-shared data must necessarily be different than that of batch shared data, because it will have only a single writer and a single reader before it is no longer needed. Pipeline-shared outputs will require some facility for discovery by the reader of the data, but need not be advertised to the same degree as batch-shared data. The loss of a pipeline-shared output may require the re-execution of a previous computation stage.

Solutions to both pipeline and batch sharing problems require that an application's I/O be classified into each of the three roles with some degree of accuracy. Custom applications such as SETI have succeeded in attaining wide scalability by virtue of manual I/O division: all endpoint I/O happens via explicit network communication. Yet, we can hardly expect that all valuable applications will be re-written for a distributed environment. Ideally, such I/O roles would be detected automatically. Such an approach is taken by the TREC [111] system, which deduces program dependencies from I/O behavior. We might also reasonably ask the user to provide hints of I/O roles to the system without modifying applications directly.

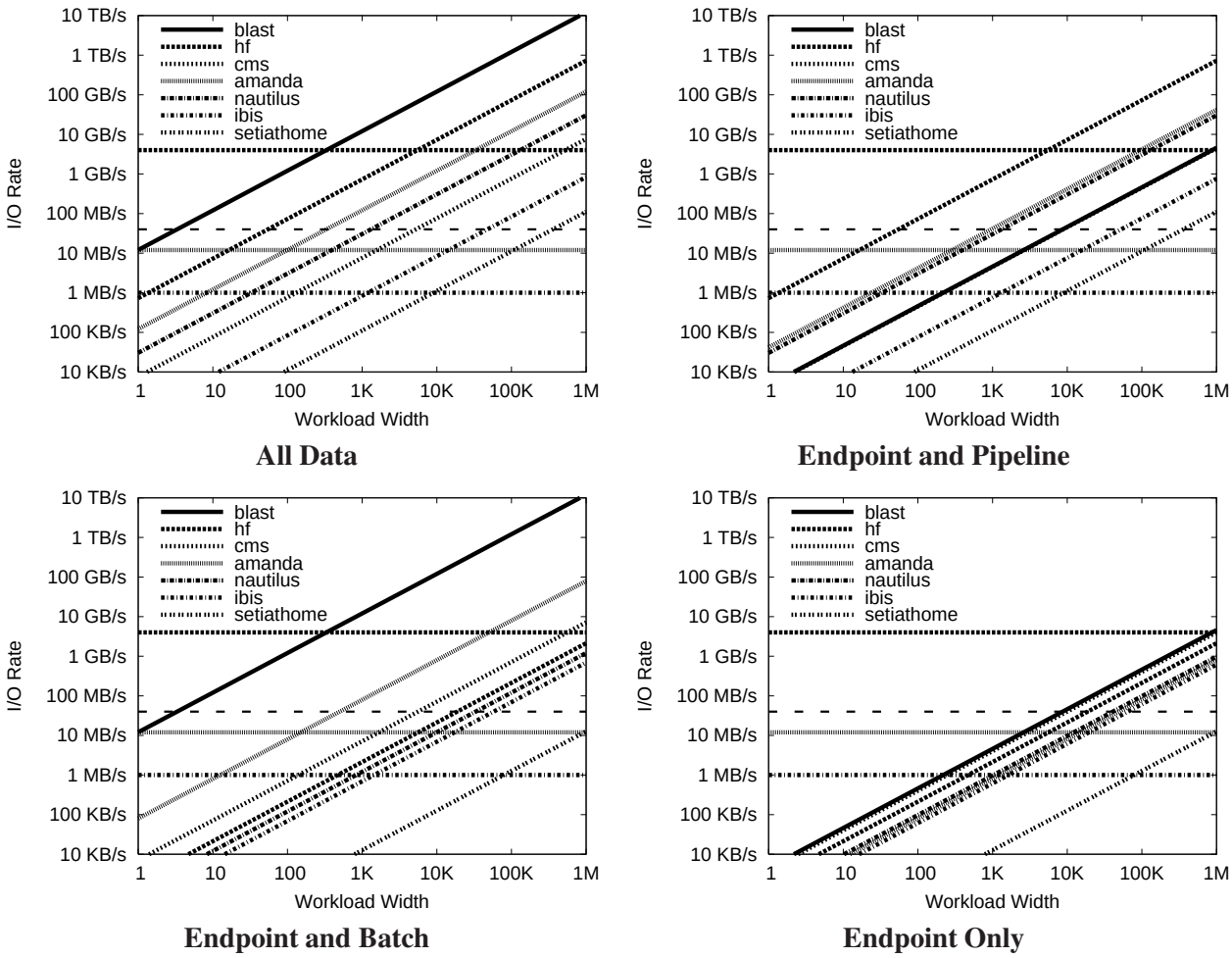


Figure 2.7: **Scalability of I/O Roles.** *These graphs show how the scalability of these applications can be improved by up to several orders of magnitude when batch-shared and pipeline-shared I/O are not performed at the endpoint server. The three horizontal lines show current milestones (as of 2005) in I/O bandwidth. The upper, at 1500 MB/s, represents a high-end storage center, the next, at 40 MB/s, represents a commodity disk, the next, at 12 MB/s, represents local-area network bandwidth, and the lowest, at 1 MB/s, represents wide-area network bandwidth.*

Appl.	CPU/IO (MIPS/MBPS)	MEM/CPU (MB/MIPS)	CPU/IO (instr/op)
seti	45888	0.15	8737 K
blastp	37	26.77	144 K
ibis	34530	0.20	109823 K
cmkin	801	0.26	6372 K
cmsim	189	1.86	393 K
total	190	2.09	396 K
setup	8	0.06	27 K
argos	311	0.02	850 K
scf	34	0.30	189 K
total	74	0.16	353 K
nautilus	4501	1.71	19496 K
bin2coord	1350	0.00	4403 K
rasmol	566	0.02	1991 K
total	2287	1.20	8238 K
corsika	6854	0.14	27670 K
corama	76	0.06	313 K
mmc	2189	0.10	310 K
amasim2	191	12.48	150443 K
total	785	3.77	551 K
Amdahl	**8**	**1.00**	**50 K**
Gray	**8**	**1 - 4**	**>50 K**

Table 2.5: **Balanced System Ratios.** *This table shows ratios between the I/O, memory, and CPU requirements of these applications. We compare our observed ratios to those originally proposed by Gene Amdahl [6] and later amended by Jim Gray [52].*

A number of file systems take account of the conventional wisdom that quickly-deleted data is a significant source of traffic in general-purpose workload. However, this recognition has limited application due to the requirements of reliability and consistency in interactive systems. For example, NFS permits a 30-60 second delay between application writes and data movement to the server. Were this delay made to be minutes or hours in order to accommodate pipeline sharing, the reduction in unnecessary writes would be accompanied by a much increased danger of data loss during a crash and some very unusual consistency semantics. The session semantics of AFS are even worse: closing a file is a blocking operation that forces the write-back of dirty data. Not only would all vertically shared data be written back at each of the (numerous) close operations, but the CPU would be held idle between pipelines, offering no possibility of CPU-I/O overlap.

General-purpose file systems operate under the assumption that most data must eventually flow back to the archival site. These workloads require the opposite assumption: most created data should remain *where it is created* until an explicit operation by the writer, the system, or perhaps the user forces it into archival storage. This improves overlap and eliminates unnecessary writes, but increases the danger that I/O operations waiting to be written back may fail due to permissions, disconnection, or any of the many other sources of error in a distributed file system. This is acceptable in a batch system, as long as such a failed I/O can be detected, matched with the process that issued it, and a re-execution of the job can be forced.

One approach which we will present in detail in Chapter 4 is coupling this I/O information with a workflow manager, such as Condor's DAGMan or Globus' Chimera [46], both of which already track the dependencies in general graphs of jobs. In both of these systems, I/O activity is presumed to be a reliable (and centralized) side effect of execution. However, if the creation and positioning of workload data is integrated into the workflow, such data can be efficiently shared while still maintaining the possibility of error recovery.

2.6 Discussion

Applications are not run in isolation. In production settings, scripting and workflow tools are used to glue together series of applications into pipelines; a particular pipeline may be run many thousands of times over varied inputs to achieve the goals of the users. We term such workloads batch-pipelined, as batches of pipelines are run at a given instant.

In this chapter, we characterize a collection of scientific batch workloads. Beyond typical quantitative characterizations of processing, memory, and I/O demands, we bring forth the more general qualitative nature of these workloads and organize them into a taxonomy which we term batch-pipeline. By characterizing and defining this structure, we allow users and batch scheduling systems the ability to more carefully reason about and plan for these workloads.

The key to managing these workloads is I/O classification. By segregating I/O traffic by type, and through aggressively exploiting sharing characteristics, performance can be improved by many orders of magnitude as we will now show in Chapter 3.

Chapter 3

Distributed File Systems for Batch-Pipeline Workloads

Traditional distributed file systems, such as NFS and AFS, have evolved on a solid foundation of empirical measurement. By studying expected workload patterns [13, 78, 89, 96, 114], researchers and developers have long been able to make appropriate trade-offs in system design, thereby building systems that work well for the workloads of interest. Most previous distributed file systems have been targeted at a particular computing environment, namely a collection of interactively used client machines. However, as past work has demonstrated, different workloads lead to different designs (*e.g.*, FileNet [36] and the Google File System [50]); if assumptions about usage patterns, sharing characteristics, or other aspects of the workload change, one must reexamine the design decisions embedded within distributed file systems.

Having conducted our own measurement study of batch-pipeline workloads as described in Chapter 2, we are now ready to evaluate whether the design decisions within current distributed file systems are well-suited for batch-pipeline workloads. Specifically, we consider the challenge of scheduling batch-pipeline workloads in remote environments in which storage may be scarce and the wide-area connection to the home storage server may be prohibitively slow.

Batch workloads are typically run in controlled local-area cluster environments [69, 119]. However, organizations that have large workload demands increasingly need ways to share resources across the wide-area, both to lower costs and to increase productivity. One approach to accessing resources across the wide-area is to simply run a local-area batch system across multiple clusters that are spread over the wide-area and to use a distributed file system as a backplane for data access.

Unfortunately, this approach is fraught with difficulty, largely due to the way in which I/O is handled. The primary problem with using a traditional distributed file system is in its approach to *control*: many decisions concerning caching, consistency, and fault tolerance are made *implicitly* within the file system. Although these decisions are reasonable for the workloads for which these file systems were designed, they are ill-suited for a wide-area batch computing system. For example, to minimize data movement across the wide-area, the system must carefully use the cache space of remote clusters; however, caching decisions are buried deep within distributed file systems, thus preventing such control.

To mitigate these problems and enable the utilization of remote clusters for I/O-intensive batch workloads, we introduce the Batch-Aware Distributed File System (BAD-FS). BAD-FS differs from traditional distributed file

systems in its approach to control; BAD-FS exposes decisions commonly hidden inside of a distributed file system to an external workload-savvy scheduler. BAD-FS leaves all consistency, caching, and replication decisions to this scheduler, thus enabling *explicit* and workload-specific control of file system behavior.

The main reason to migrate control from the file system to the scheduler is *information* – the scheduler has intimate knowledge of the workload that is running and can exploit that knowledge to improve performance and streamline failure handling. The combination of workload information and explicit control of the file system leads to three distinct benefits over traditional approaches:

• **Enhanced performance.** By carefully managing remote cluster disk caches in a cooperative fashion, and by controlling I/O such that only needed data is transported across the wide-area, BAD-FS minimizes wide-area traffic and improves throughput. Using workload knowledge, BAD-FS further improves performance by using capacity-aware scheduling to avoid thrashing.

• **Improved failure handling.** Using basic workload information, the scheduler can determine whether to make replicas of data based on the cost of generating that data, and not indiscriminately as is typical in many file systems. Data loss is therefore treated uniformly as a performance problem. The scheduler has the ability to regenerate a lost file by rerunning the application that generated it and hence only replicates when the cost of regeneration is high.

• **Simplified implementation.** Detailed workload information allows a simpler implementation. For example, BAD-FS provides a cooperative cache but does not implement a cache consistency protocol. Through exact knowledge of data dependencies, it is the scheduler that ensures proper access ordering among jobs. Previous work has demonstrated the difficulties of building a more general cooperative caching scheme [8, 23].

We demonstrate the benefits of explicit control via our prototype implementation of BAD-FS. Using synthetic workloads, we demonstrate that BAD-FS can reduce wide-area I/O traffic by an order of magnitude, can avoid performance faults through capacity-aware scheduling, and can proactively replicate data to obtain high performance in spite of remote failure. Using real workloads, we demonstrate the practical benefits of our system: I/O-intensive batch workloads can be run upon remote resources both easily and with high performance.

Finally, BAD-FS achieves these ends while maintaining site autonomy and support for unmodified legacy applications. Both of these practical constraints are important for acceptance in wide-area batch computing environments.

In the remainder of this chapter, we describe assumptions about the expected environment and workload, discuss the architecture of our system, present our experimental evaluation, and end this chapter with some summary observations about the need for a new batch-aware distributed file system that exports storage control to a workload-savvy scheduler.

3.1 Architecture

In this section, we present the architecture and implementation of BAD-FS. The main goal of the design of BAD-FS is to export sufficient control to a remote scheduler. This external control allows the batch scheduler to deliver improved performance and better fault-handling for I/O-intensive batch-pipeline workloads run on remote clusters.

BAD-FS is structured as is shown in Figure 3.1. Two types of server processes manage local resources. A *compute server* exports the ability to transfer and execute an ordinary user program on a remote CPU. A *storage*

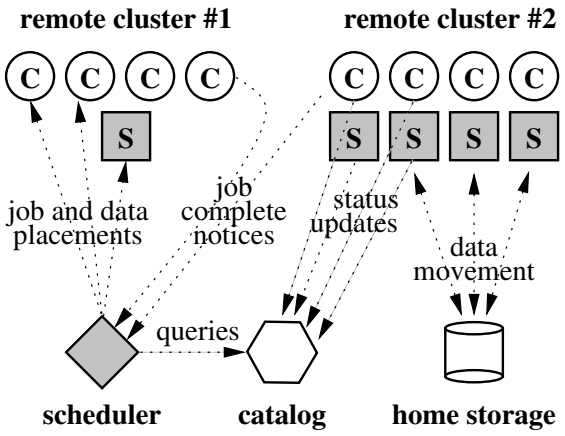


Figure 3.1: **System Architecture.** *This figure summarizes the architecture of BAD-FS, with the novel elements shaded gray. Circles are compute servers, which execute batch jobs. Squares are storage servers, which hold cached inputs and temporary outputs. Both types of servers report to a catalog server, which records the state of the system. The scheduler uses information from the catalog to direct the system by configuring storage devices and submitting batch jobs. The gray shapes are novel elements in our design; the white are standard components found in batch scheduling systems.*

server exports access to disk and memory resources via remote procedure calls that resemble standard file system operations. It also permits remote users to allocate space via an abstraction called *volumes*. *Interposition agents* bind unmodified workloads running on compute servers to storage servers. Both types of servers periodically report themselves to a *catalog server*, which summarizes the current state of the system. A *scheduler* periodically examines the state of the catalog, considers the work to be done, and assigns jobs to compute servers and data to storage servers. The scheduler may obtain data, executables, and inputs from any number of external storage sites. For simplicity, we assume the user has all the necessary data stored at a single *home storage server* such as a standard FTP server.

From the perspective of the scheduler, compute and storage servers are logically independent. A specialized device might run only one type of server process: for example, a diskless workstation runs only a compute server, whereas a storage appliance runs only a storage server. However, a typical workstation or cluster node has both computing and disk resources and thus runs both.

BAD-FS may be run in an environment with multiple owners and a high failure rate. In addition to the usual network and system errors, BAD-FS must be prepared to handle *eviction* failures in which shared resources may be revoked without warning. An additional challenge is that the rapid rate of change in such systems creates possibly stale information in the catalog. BAD-FS must also be prepared to discover that the servers it attempts to harness may no longer be available.

The BAD-FS implementation makes use of several standard components. Namely, the compute servers are Condor [69] *startd* processes, the storage servers are modified NeST storage appliances [18], the interposition agents are Parrot [107] agents, and the catalog is the Condor *matchmaker*. The servers advertise themselves to the catalog via the ClassAd [85] resource description language.

3.1.1 Storage Servers

Storage servers are responsible for exporting the raw storage of the remote sites in a manner that allows efficient management by remote schedulers. A storage server does not have a fixed policy for managing its space. Rather,

it makes several policies accessible to external users who may carve up the available space for caching, buffering, or other tasks as they see fit. Using an abstraction called *volumes*, storage servers allow users to allocate space with a name, a lifetime, and a type that specifies the policy by which to internally manage the space. The BAD-FS storage server exports two distinct volume types: scratch volumes and cache volumes.

A *scratch volume* is a self-contained read-write file system, typically used to localize access to temporary data. The scheduler can use scratch volumes for pipeline data passed between jobs and as a buffer for endpoint output. Using scratch volumes, the scheduler minimizes home server traffic by localizing pipeline I/O and only writing endpoint data when a pipeline successfully completes. To permit efficient backup, a storage server can be directed to duplicate a scratch volume onto another server.

A *cache volume* is a read-only view of a home server, created by specifying the name of the home server and path, a caching policy (*i.e.*, LRU or MRU), and a maximum storage size. Multiple cache volumes can be bound into a *cooperative cache volume* by specifying the name of a catalog server, which the storage servers query to discover their peers. A number of algorithms [33, 38] exist for managing a cooperative cache, but it is not our intent to explore the range of these algorithms here. Rather, we describe a reasonable algorithm for this system and explain how it is used by the scheduler.

Cooperative disk cache

The cooperative cache is built using a distributed hash table [54, 68]. The keys in the table are block addresses, and the values specify which server is primarily responsible for that block. To avoid wide-area traffic, only the primary server will fetch a block from the home server and the other servers will create secondary copies from the primary (*i.e.* the nodes self-organize into a multi-cast tree of depth two). When space is needed, secondary data is evicted before primary. To approximate locality, our initial implementation only forms cooperative caches between peers in the same subnetwork. We present a brief discussion of other alternatives later in this dissertation in Chapter 6.

Failures within the cooperative cache, including partitions, are easily managed but may cause slowdown. Should a cooperative cache be internally partitioned, the primary blocks that were assigned to the now missing peers will be reassigned. As long as the home server is accessible, partitioned cooperative caches will be able to refetch any lost data and continue without any noticeable disturbance (beyond runtime perturbation) to running jobs.

This approach to cooperative caching has two important differences from previous work. First, because data dependencies are completely specified by the scheduler, we do not need to implement a cache consistency scheme. Once read, all data are considered current until the scheduler invalidates the volume. This design decision greatly simplifies our implementation; previous work has demonstrated the many difficulties of building a more general cooperative caching scheme [8, 23]. Second, unlike previous cooperative caching schemes that manage cluster memory [33, 38], our cooperative cache stores data on local *disks*. Although managing memory caches cooperatively could also be advantageous, the most important optimization to make in our environment is to avoid data movement across the wide-area; managing remote disk caches is the simplest and most effective way to do so.

```
job       a     a.condor
job       b     b.condor
job       c     c.condor
job       d     d.condor
parent    a     child    b
parent    c     child    d
volume    b1    ftp://home/data 1 GB
volume    p1    scratch 50 MB
volume    p2    scratch 50 MB
mount     b1    a     /mydata
mount     b1    c     /mydata
mount     p1    a     /tmp
mount     p1    b     /tmp
mount     p2    c     /tmp
mount     p2    d     /tmp
extract   p1    x     ftp://home/out.1
extract   p2    x     ftp://home/out.2
```

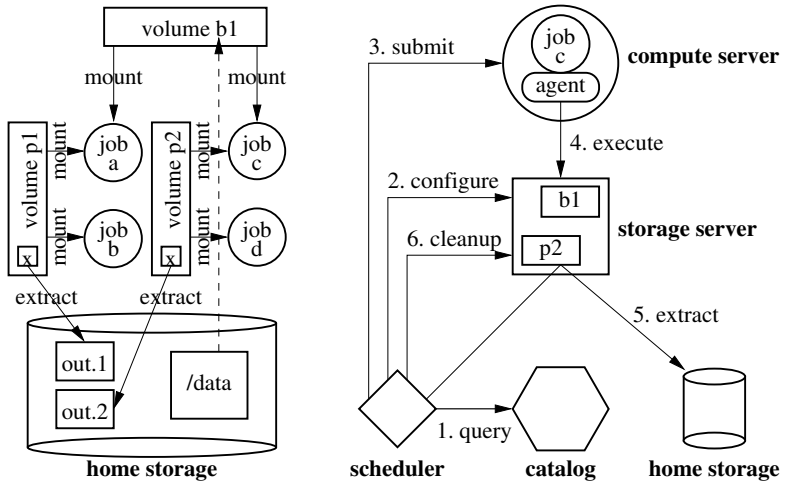


Figure 3.2: **Workflow and Scheduler Examples.** *Shown here is an example of the declarative workflow language that describes a batch-pipelined workload in (A), the logical representation of this workload in (B), and the execution plan formulated by the scheduler from this workload in (C).*

Local vs. Global Control

Note that volumes export only a certain degree of control to the scheduler. Namely, by creating and deleting volumes, the scheduler controls which data sets reside in the remote cluster. However, the storage servers retain control over per-block decisions. Two such important decisions made locally by the storage servers are the assignment of primary blocks in the cooperative cache and cache victim selection. Of course, if the scheduler is careful in space allocation, the cache will only victimize blocks that are no longer needed. In general, we have found this separation of global and local control to be suitable for our workloads. Although more work needs to be done to precisely identify the balance point, it is clear that a trade-off is better than either extreme. Complete local control, the current approach, suffers because the policies embedded within distributed file systems are inappropriate for batch workloads. The other extreme, complete global control, in which the scheduler makes decisions for each block of data, would require exorbitant complexity in the scheduler and would incur excessive network traffic to exert this fine-grained control.

3.1.2 Interposition Agents

In order to permit ordinary workloads to make use of storage servers, an *interposition agent* [58] transforms POSIX I/O operations into storage server calls. The agent's mapping from logical path names to physical storage volumes is provided by the scheduler at runtime. Together, the agent and the volume abstraction can hide a large number of errors from the job and the end user. For example, if a volume no longer exists, whether due to accidental failure or deliberate preemption, a storage server returns a unique *volume lost* error to the agent. Upon discovering this, the agent forcibly terminates the job, indicating that it could not run correctly in the given environment. This gives the scheduler clear indication of failures and allows it to take transparent recovery actions.

3.1.3 The Scheduler

The BAD-FS scheduler directs the execution of a workload on compute and storage servers by combining a static workload description with dynamic knowledge of the system state. Specifically, the scheduler minimizes traffic across the wide-area by differentiating I/O types and treating each appropriately, carefully managing remote storage to avoid thrashing and replicating output data proactively if that data is expensive to regenerate.

Workflow language

Figure 3.2-A shows a simple workflow script in which the keyword `job` names a job and binds it to a description file, which specifies the information needed to execute that job. `Parent` indicates an ordering between two jobs. The `volume` keyword names the data sources required by the workload. For example, volume `b1` comes from an FTP server, while volumes `p1` and `p2` are empty scratch volumes. Volume sizes are provided to allow the scheduler to allocate space appropriately. The `mount` keyword binds a volume into a job's namespace. For example, jobs `a` and `c` access volume `b1` as `/mydata`, while jobs `a` and `b` share volume `p1` via the path `/tmp`. The `extract` command indicates which files of interest must be committed to the home storage server (*i.e.*which are endpoint outputs). In this case, each pipeline produces a file `x` that must be retrieved and uniquely renamed.

Figure 3.2-B shows the graphical presentation of this workload and maps to the scheduler's internal data structures used to represent it. In Figure 3.2-C is the scheduler's plan for executing job c:

1. The scheduler queries the catalog for the current system state and decides where to place job c and its data.
2. The scheduler creates volumes b1 and p2 on a storage server.
3. Job c is dispatched to the compute server.
4. Job c executes, accessing its volumes via the agent.
5. After jobs c and d complete, the scheduler extracts x from p2.
6. The scheduler frees volumes b1 and p2.

I/O Scoping

Unlike most file systems, BAD-FS is aware of the flow of its data. From the workflow language, the scheduler knows where data originates and where it will be needed. This allows it to create a customized environment for each job and minimize traffic to the home server. We refer to this as *I/O scoping*.

I/O scoping minimizes traffic in two ways. First, cooperative cache volumes are used to hold read-only batch data such as `b1` in Figure 3.2-A. Such volumes may be reused without modification by a large number of jobs. Second, scratch volumes, such as `p2` in Figure 3.2-A, are used to localize pipeline data. As a job executes, it accesses only those volumes that were explicitly created for it; the home storage server is accessed only once for batch data and not at all for pipeline.

Consistency management

With the workload information expressed in the workflow language, the scheduler neatly addresses the issue of consistency management. All of the required dependencies between jobs and data are specified directly. Since the scheduler only runs jobs so as to meet these constraints, there is no need to implement a cache consistency protocol among the BAD-FS storage servers.

The user may make mistakes in the workflow description that can affect both cache consistency and correct failure recovery. However, through an understanding of the expected workload behavior as specified by the user, the scheduler can easily detect these mistakes and warn the user that the results of the workload may have been compromised. We have not implemented these detection features, but the architecture readily admits them.

Capacity-Aware Scheduling

The scheduler is responsible for throttling a running workload to avoid performance faults and maximize throughput. By carefully allocating volumes, the scheduler avoids overflowing storage or thrashing caches. Although disk capacity is rapidly increasing, the size of data sets is also growing and space management remains important [8, 43, 53, 56].

The scheduler manages space by retrieving a list of available storage from the catalog server and selecting the ready job with the least unfulfilled storage needs, whether pipe or batch. If the scheduler is able to allocate all of that job's volumes, then it allocates and configures these volumes and schedules the job. If there are no jobs to execute or not enough available space, then the scheduler waits for a job to complete, more resources to arrive, or for a failure to occur. Note that due to a lack of complete global control, the scheduler may need to slightly overprovision when the needed volume size approaches the storage capacity.

In other scheduling domains, selecting the smallest job first can result in starvation. In this domain, however, starvation is avoided because a workflow is a static entity executed by one scheduler. Although smaller jobs will run first, all jobs will eventually be run. Multiple workflows or multiple users are represented by multiple schedulers, each operating independently. At that level, an entire workflow could starve another, but that is beyond our ability to control. We briefly discuss additional scheduling challenges presented by multiple workloads in Chapter 6.

Failure Handling

Finally, the scheduler makes BAD-FS robust to failures by handling failures of jobs, storage servers, the catalog, and itself. One aspect of batch workloads that we leverage is job idempotency; a job can simply be rerun in order to regenerate its output.

The scheduler contains hard-wired logic by which it waits for passive indications of failure in compute and storage servers and then conducts active probes to verify. For example, if a job exits abnormally with an error indicating a failure detected by the interposition agent, then the scheduler suspects that the storage servers housing one or more of the volumes assigned to the job are faulty. The scheduler then probes the servers. If all volumes are healthy, it assumes the job encountered transient communication problems and simply reruns it. However, if the volumes have failed or are unreachable for some period of time, they are assumed lost.

The failure of a volume affects the jobs that use it. As a design simplification, the scheduler considers a partial volume failure to be a failure of the entire volume. Running jobs that rely on a failed volume must be stopped. In addition, failures can cascade; completed processes that *wrote* to a volume must be rolled back and re-run. In order to avoid these expensive restarts of a pipeline, the scheduler may checkpoint scratch volumes as pipeline stages complete.

Of course, determining an optimal checkpoint interval is an old problem [48]. The solution depends upon the likelihood of failure, the value of a checkpoint, and the cost to create it. Unlike most systems, BAD-FS can solve this problem automatically, because the scheduler is in a unique position to measure the controlling variables. The

scheduler performs a simple cost-benefit analysis at run-time to determine if a checkpoint is worthwhile.

The algorithm works as follows. The scheduler tracks the average time to replicate a scratch volume. This cost is initially assumed to be zero in order to trigger at least one replication and measurement. To determine the benefit of replication, the scheduler tracks the number of job and storage failures and computes the mean-time-to-failure across all devices in the system.

The benefit of replicating a volume is the sum of the run times of those jobs completed so far in the applicable pipeline multiplied by the probability of failure. If the benefit exceeds the cost, then the scheduler replicates the volume on another storage server as insurance against failure. When the original fails, the scheduler can restart the pipeline using the saved copy.

Due to its robust failure semantics, the scheduler need not handle network partitions any differently than other failures. When partitions are formed between the scheduler and compute servers, the scheduler may choose to reschedule any jobs that were running on the other side of the partition. In such a situation, it is possible that the partition could be resolved, at which point the scheduler will find that multiple servers are executing the same jobs. Note that this will not introduce errors because each job writes to distinct scratch volumes. The scheduler may choose one output to extract and then discard the other.

3.1.4 Practical Issues

One of the primary obstacles to deploying a new distributed system is the need for a friendly administrator. Whether deploying an operating system, a file system, or a batch system, the vast majority of such software requires a privileged user to install and oversee the software. Such requirements make many forms of distributed computing a practical impossibility; the larger and more powerful the facility, the more difficult it is for an ordinary user to obtain administrative privileges. To this end, BAD-FS is packaged as a *virtual batch system* that can be deployed over an existing batch system without special privileges. This technique is patterned after the "glide-in job" described by Frey *et al.* [47] and is similar in spirit to recursive virtual machines [42].

To run BAD-FS, an ordinary user need only to be able to submit jobs into an existing batch system. BAD-FS bootstraps itself on these systems, relying on the basic ability to queue and run a self-extracting executable program containing the storage and compute servers and the interposition agent. Once deployed, the servers report to a catalog server, and the scheduler may then harness their resources. Note that the scheduling of the virtual batch jobs is at the discretion of the host system; these jobs may be interleaved in time and space with jobs submitted by other users. We have used this technique to deploy BAD-FS over several existing Condor and PBS batch systems.

Another practical issue is security. BAD-FS currently uses the Grid Security Infrastructure (GSI) [44], a public key system that delegates authority to remote processes through the use of time-limited proxy certificates. To bootstrap the system, the submitting user must enter a password to unlock the private key at his/her home node and generate a proxy certificate with a user-settable timeout. The proxy certificate is delegated to the remote system and used by the storage servers to authenticate back to the home storage server. This requires that users trust the host system not to steal their secrets, which is reasonable in a c2c environment.

3.2 User Burden

To many readers accustomed to working in an interactive environment, specifying this degree of workload information may seem like an unusual burden. We point out that a user intending to execute batch-pipelined

workloads must be exceptionally organized. Batch users already provide this information, but it is scattered across shell scripts, make files, and batch submission files. In addition to imparting needed information to the BAD-FS scheduler, the workflow language actually reduces user burden by collecting all of this dispersed information into a coherent whole.

Although user burden can be reduced in this way, whether the resulting burden is acceptable to the batch user remains to be seen. While users should be able to properly characterize the different data types within their workloads, a further unanswered question is their ability to accurately quantity the amounts of data within each type. Although more studies are necessary to properly answer this question, valid arguments can be made for both possible answers.

Batch users should be both able and willing to provide basic accurate workload information for several reasons. First, the "typical" batch user is likely to be more technically adept than an interactive one and therefore more easily able to observe, profile and describe the I/O behavior of their workloads. Further, our target workloads are submitted by "super-users" who submit workloads with batch-widths into the hundreds and thousands. This massive amount of work suggests both that these users have multiple opportunities to observe and profile their workloads as well as an increased incentive to do so.

Further, the number of degrees of separation between batch users and the developers of batch software is likely fewer than that between interactive users and the developers of interactive software. Batch users are much more likely to either have developed their own software or to have inherited it from a colleague. As such, these users have an advantageous perspective which allows introspection into the behavior of their workloads.

However, for several reasons it is also likely that users may struggle to provide accurate and precise workload information. Previous studies of runtime prediction have found that user estimates are rather poor [27, 65, 75]. As such, their ability to provide I/O predictions is likely to be similarly poor or even poorer as I/O predictions are arguably more difficult than runtimes. Second, the number of degrees of separation between users and developers in batch systems, while fewer than in interactive, is growing. More users are using inherited programs which may have seen multiple iterations of inheritance; also, the use of proprietary programs such as BLAST is frequent.

Finally, even should users be able to provide this information, it is clear that they would almost always prefer not to. These users would prefer to concentrate on the details of their own studies blissfully ignorant of the internal workings of the batch scheduling system.

3.3 Experimental Evaluation

In this section, we present an experimental evaluation of BAD-FS under a variety of workloads. We first present our methodology, and then focus on I/O scoping, capacity-aware scheduling, and failure handling, using synthetic workloads to understand system behavior. Second, we present our experience running real workloads on our system in a controlled environment. Finally, we discuss our initial experience using BAD-FS to run real workloads across multiple clusters in the wild.

3.3.1 Methodology

In the initial experiments in this section, we construct an environment in which we assume the user's input data is stored on a home server and the user has access to a remote compute cluster. Once all pipelines have run and all output data is safely stored back at the home server, the workload is considered complete.

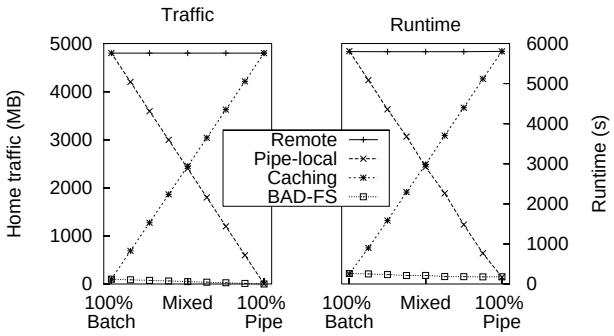


Figure 3.3: **I/O Scoping: Traffic Reduction and Run Times.**

We assume that the workload is run on a remote cluster of machines, accessible from the user's home server via a wide-area link. To emulate this scenario, we limit the bandwidth to the home server to 1 MB/s via a simple network delay engine similar to DummyNet [87]. Thus, all I/O between the remotely run jobs and the home server must traverse this slow link. The cluster itself is taken from a dedicated compute pool of Condor nodes at the University of Wisconsin, connected via a 100 Mbit/s Ethernet switch. Each node has two Pentium-3 processors, 1 GB of physical memory and a 9 GB IBM SCSI drive, of which only a 1 GB partition is made available to Condor jobs. Of these 1 GB partitions, typically only about half is available at any one time as the rest awaits lazy garbage collection.

To explore the performance of BAD-FS under a range of workload scenarios, we utilize a parameterized synthetic batch-pipeline workload. The synthetic workload can be configured to perform varying amounts of endpoint, batch, and pipeline I/O, compute for different lengths of time, and can exhibit different amounts of both batch and pipeline parallelism. As each experiment requires different parameters, we leave those descriptions for the individual figure captions. However, given our previous results in workload analysis as described previously in this dissertation in Chapter 2, we focus on *batch-intensive* workloads, which exhibit a high degree of batch sharing but little pipeline or endpoint I/O, and *pipe-intensive*, which perform large amounts of pipeline I/O but generate little batch or endpoint I/O. Note that *endpoint-intensive* workloads are not discussed as these workloads would not benefit from BAD-FS. Rather, these workloads would be well-served either through remote I/O or through a more explicit, user-defined, data movements as in the Stork system [60].

3.3.2 I/O Scoping

The results of the first experiment, as shown in Figure 3.3, demonstrate how BAD-FS uses I/O scoping to minimize traffic across the wide area by localizing pipeline I/O in scratch volumes and reusing batch data in cooperative cache volumes. These graphs show the total amount of network traffic generated by and runtimes for a number of different workloads with different optimizations enabled. For this experiment, we run 48 synthetic pipelines of depth 4, each of which generates a total of 100 MB I/O. Across the x-axis we vary the relative amounts of batch I/O and pipeline I/O. For example, at 100% Batch, the workload generates 100 MB of batch I/O and no pipeline. As is common in these types of workloads, the amount of endpoint I/O is small (1 KB). The leftmost graph shows the total amount of home server traffic; the right shows total runtimes when the home server is accessed over an emulated wide-area network (set at 1 MB/s).

Although these optimizations are straightforward, their ability to increase throughput is significant. In this

experiment, we repeatedly run the same synthetic workload but vary the relative amount of batch and pipeline I/O. We compare a number of different system configurations. In the *remote* configuration, all I/O is sent to the home node. Against this baseline, we compare the *pipeline localization* and *caching* optimizations. Finally, both optimizations are combined in the BAD-FS configuration. Note that in these experiments, we assume copious cache space and a controlled environment; neither capacity-aware scheduling nor failure recovery is needed.

The left-hand graph shows the total I/O that is transferred over the wide-area network. Not surprisingly, the cooperative cache greatly reduces batch traffic to the home node by ensuring that all but the first reference to a batch data set is retrieved from the cache. We can also see that the pipeline localization optimizations work as expected, removing pipeline I/O entirely from the home server. Finally, we see that neither optimization in isolation is sufficient; only the BAD-FS configuration that combines both is able to minimize network traffic throughout the entire workload range. The right-hand graph in Figure 3.3 shows the runtimes of the workloads on our emulated remote cluster. From this graph, we can see the direct impact that wide-area traffic has on runtime.

3.3.3 Capacity-Aware Scheduling

Next, we examine the benefits of explicit storage management. The previous experiments were run in an environment where storage was not used to near capacity. With the increasing size of batch data sets and storage sharing by jobs and users, the scheduler must carefully manage remote space so as to avoid over-allocations of storage. Storage must be carefully allocated because an over-allocation of batch data can lead to wide-area thrashing as data which is redundantly accessed must be redundantly fetched from the home storage server. An over-allocation of pipeline data, on the other hand, can cause write failures.

For these experiments, we compare the capacity-aware BAD-FS scheduler to two simple variants: a *depth-first* scheduler and a *breadth-first* scheduler. These algorithms are not aware of the data needs of the workload and base decisions solely on the job structure within it. Depth-first simply assigns a single pipeline to each available CPU and runs all jobs in the pipeline to completion before starting another. Conversely, breadth-first attempts to execute all jobs at a particular depth before descending to the next horizontal slice of the workload.

Each type of traversal may be correct for certain types of workloads, but can lead to poor storage allocations in others. For example, depth-first scheduling of a batch-intensive workload is more likely to cause thrashing because it attempts to simultaneously cache all of the batch datasets. Similarly, breadth-first scheduling of a pipe-intensive workload is more likely to over-allocate storage because it creates allocations for all pipelines before completing any.

3.3.4 Batch-intensive Capacity-Aware Scheduling

Figure 3.4 illustrates the importance of capacity-aware scheduling through measurements of batch-intensive workloads scheduled using various algorithms. We can make a number of observations from these graphs. First, the similarity between the graphs validates that the wide-area network link is an important bottleneck resource. Second, as expected, the different policies achieve similar results as long as the entirety of all four batch data sets fits within the caches (*i.e.*, up to 25%). As the size of the batch data approaches the total capacity of the cooperative cache, the runtime and wide-area traffic increase for depth-first scheduling. As the total batch data no longer fits in cache, depth-first scheduling must refetch batch data for each pipeline. In this case, this results in three extra fetches because with 64 pipelines and 16 compute servers, each server executes four pipelines.

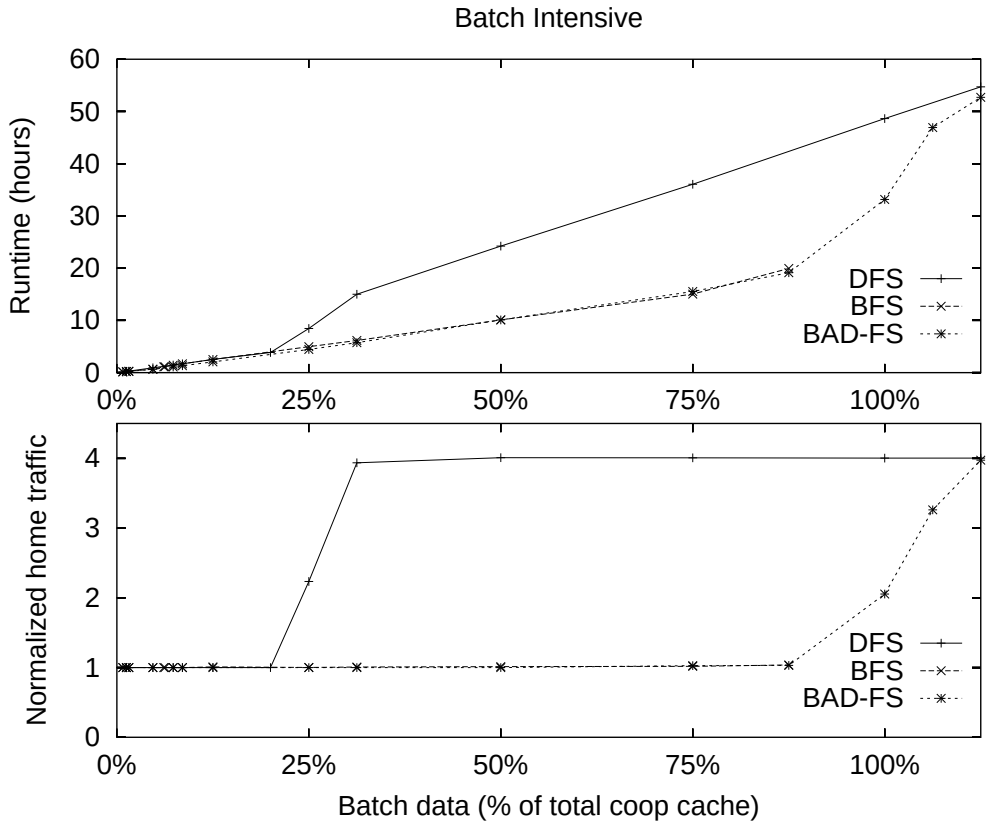


Figure 3.4: **Batch-intensive Explicit Storage Management.** *These graphs show the benefits of explicit storage management under a batch-intensive workload. The workload consists of 32 4-stage pipelines; within each stage, each process streams through a shared batch file (i.e., there are 4 batch files total). Batch file size is varied as a percentage of the total amount of cooperative cache space available across the 16 nodes in the experiment. All other I/O amounts are negligible. Each of 16 nodes has local storage which is used as a portion of the cache. The total cache size available is set to 8 GB (100% on the x-axis), which reflects our observations of available storage in the UW Condor pool. The upper graph shows the runtime and the lower presents the amount of wide-area traffic generated, normalized to the size of the batch data.*

Third, note that the runtime actually begins to increase slightly before 25%. The reason for this inefficiency is the lack of complete global control allowed through the current volume interface. In this case, the local cooperative cache hash function is not perfectly distributing data across its peers; when the cache nears full utilization, this skew overloads some nodes and results in extra traffic to the home server. Because we believe that this trade-off between local and global control is correct, the implication here is that the scheduler must be aware not only of the overall utilization of the cooperative cache, but also of the utilization of each peer.

Finally, breadth-first and BAD-FS scheduling are able to retain linear performance in this regime because they ensure that the total amount of batch data accessed at any one time does not exceed the capacity of the cooperative cache. However, once each individual batch dataset exceeds the capacity of the cooperative cache, the performance of breadth-first and BAD-FS scheduling converges with that of depth-first. Note that the same inefficiency that caused depth-first to deviate slightly before 25% causes this to happen slightly before 100%.

Finally, even in the regime in which all of the batch data fits within the cooperative cache, BAD-FS slightly outperforms both breadth- and depth-first scheduling. Due to its explicit control of the caches, the BAD-FS scheduler is able to control the replacement policy of the caches. By explicitly flushing old batch data from the storage servers as it descends through the workload, the scheduler increases their hit rates by freeing more room in the caches. In this regime in which the batch data is striped across the cooperative cache, doing so does not reduce the

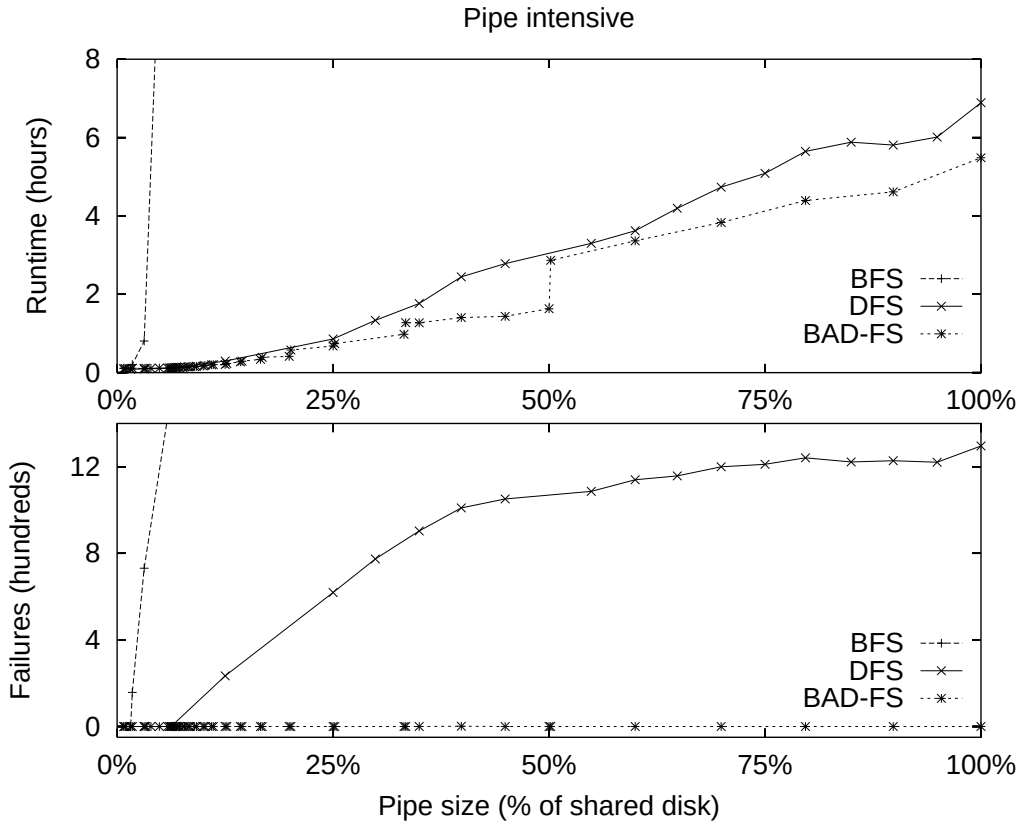


Figure 3.5: **Pipe-intensive Explicit Storage Management.** *These graphs depict the benefits of explicit storage management under a pipe-intensive workload. The workload consists of 32 4-stage pipelines, and pipe data size is varied as a percent of total storage available. All other I/O amounts are negligible. There are 16 compute servers and 1 storage server in this experiment (representing a set of diskless clients and a single server). The storage space at the server is constrained to 512 MB. The upper graph shows the runtime and the lower presents the number of failed jobs induced by each strategy.*

amount of traffic to the home node, but it does increase the amount of cache space each storage server has available for secondary data and thereby reduces the amount of local area traffic within the cooperative cache. A secondary reason for this improvement is that the pure breadth-first scheduler waits for all processes in one batch to complete before scheduling the next; the BAD-FS scheduler instead will begin the execution of the processes in the next stage of the pipeline if there is room for their data in the cache, thus improving machine utilization and increasing throughput.

3.3.5 Pipe-intensive Capacity-Aware Scheduling

In our next set of cache management experiments, we focus on a pipeline-intensive workload instead of a batch-intensive one. In this case, we expect the capacity-aware approach to follow the depth-first strategy more closely. Results are presented in Figure 3.5.

In the lower graph, we plot the number of failed jobs that each strategy induces. Job failure arise in this workload when there is a shortage of space for pipeline output; in such a scenario, a job that runs out of space for pipeline data aborts and must be rerun at some later time. Hence, the number of job failures due to lack of space is a good indicator of the scheduler's success in scheduling pipeline-intensive jobs under space constraints.

From the graph, we can observe that breadth-first scheduling is unable to prevent thrashing. In contrast, the capacity-aware BAD-FS scheduler does not exceed the available space for pipelines and thus never observes an

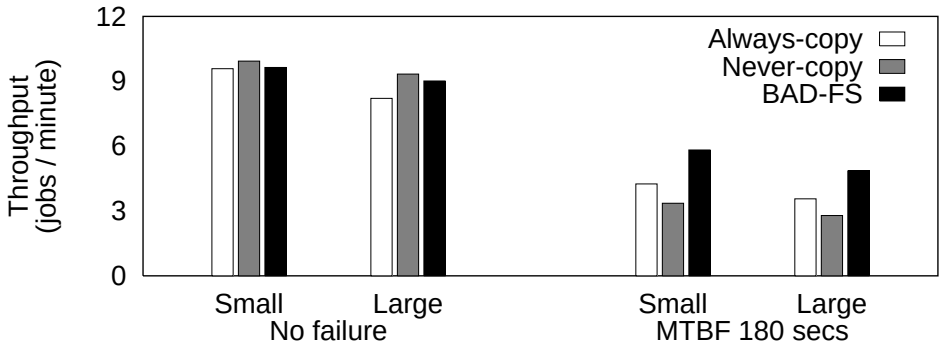


Figure 3.6: **Failure Handling.** *This graph shows the behavior of the cost-benefit strategy under different failure scenarios. Shown are two different workloads of width 64, depth 3 and one minute of CPU time; one performs a small amount of pipeline I/O, the other a large amount. Each is run both during periods of high and low rates of failure. Failures were induced using an artificial failure generator which formatted disks at random with a mean time between failures of 180 seconds, roughly corresponding to the total runtime of a single pipe.*

aborted job. This careful allocation results in a drastically reduced runtime which is shown in the upper graph.

The stair-step pattern in the runtime of BAD-FS results from this careful allocation. When the size of the data in each pipeline is between 25% and 33% of the total storage, BAD-FS schedules workload jobs on only 3 of the 16 available CPUs; between 33% and 50% on just two; and as the data exceeds 50%, BAD-FS allocates only a single CPU at a time. Notice that BAD-FS achieves runtimes comparable or better than that of depth-first scheduling without any wasted resource consumption.

3.3.6 Failure Handling

We now show the behavior of BAD-FS under varying failure conditions in Figure 3.6. Recall that unlike traditional distributed systems, the BAD-FS scheduler knows exactly how to re-create a lost output file; therefore, whether to make a replica of a file on the remote cluster should depend on the cost of generating the data versus the cost of replicating it. This choice varies with the workload and the system conditions. Figure 3.6 shows how the BAD-FS cost-benefit analysis adapts to a variety of workloads and conditions. We compare to two naive algorithms: *always-copy*, which replicates a pipeline volume after each of its stages completes and *never-copy*, which does not replicate at all.

We draw several conclusions from this graph. In an environment without failure, replication leads to excessive overhead that increases with the amount of data. In this case BAD-FS outperforms always-copy but does not quite match never-copy because of the initial replication it needs to seed its analysis. In an environment with frequent failure, it is not surprising that BAD-FS outperforms never-copy. Less intuitively, BAD-FS also outperforms always-copy. In this case, given the particulars of the workload and the failure rate, replicating is only worthwhile after the second stage; BAD-FS correctly avoids replicating after the first stage while always-copy naively replicates after all stages.

3.3.7 Workload Experience

We conclude with demonstrations of the system running real workloads. In the first demonstration as presented in Figure 3.7, we compare the runtime performance of BAD-FS to other methods of utilizing local storage resources.

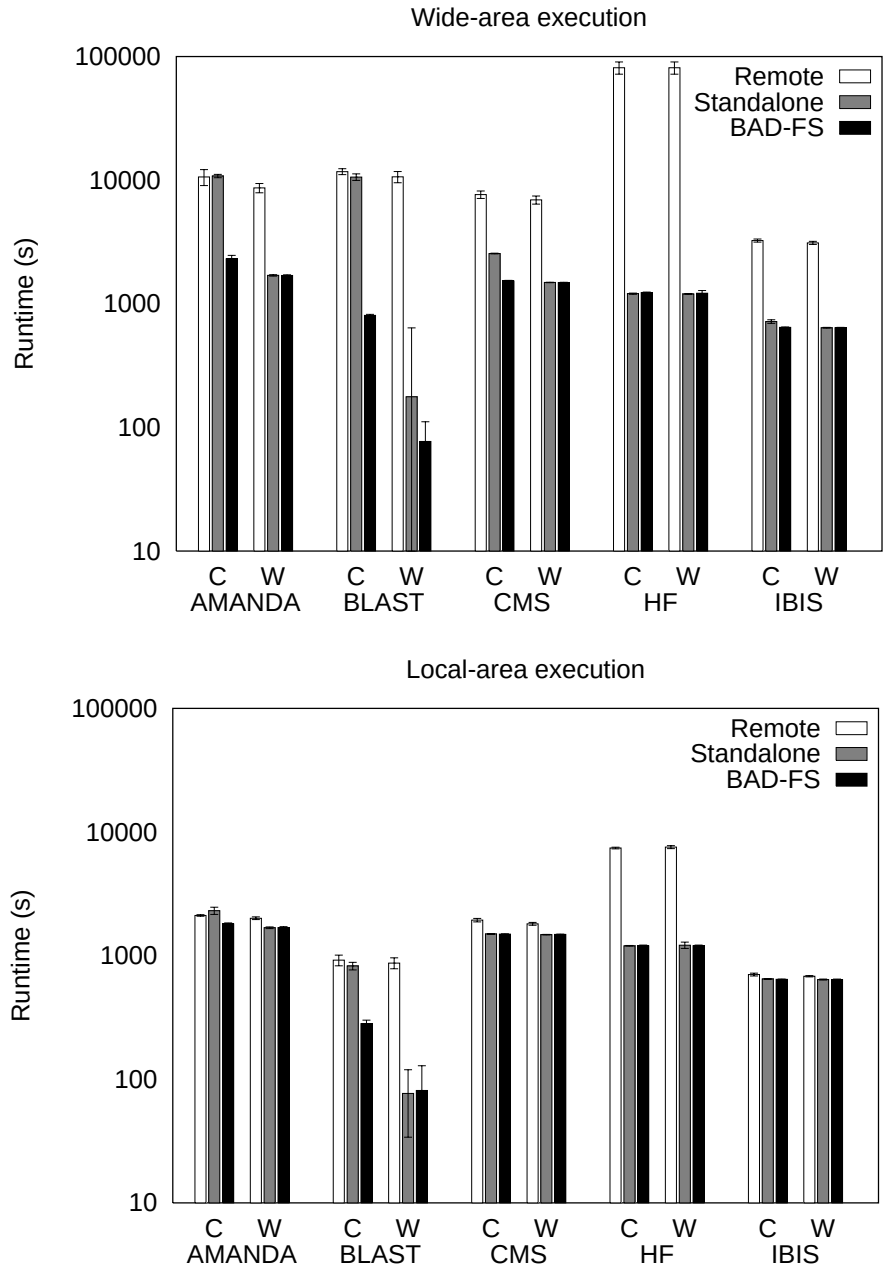


Figure 3.7: **Workload Experience.** *The upper graph shows runtimes for real jobs when the workload is executed on a cluster separated from the home node by an emulated wide-area link (set to 1 MB/s). The lower graph repeats the experiment but locates the home node within the same local area network. Note that the y-axis is shown in log scale to accentuate points of interest. For each measurement, we present average runtime for the first jobs to run on each storage server when the storage cache is* cold *(C) and for the subsequent jobs which run when the cache is* warm *(W).*

These graphs show runtime measurements of real workloads. For each workload, we submit 64 pipelines into a dedicated Condor pool of 16 CPUs. This Condor pool accesses local storage resources in one of three configurations: *remote* in which all I/O is redirected back to the home node; *standalone*, which emulates AFS-like caching to the home server; and *BAD-FS*. We were unable to actually use AFS due to a lack of adminstrative privelege to manipulate the kernels of our batch computing resources; notice this is the same problem that limits the practical deployment of many existing distributed file systems in a grid environment.

For each measurement, we present average runtime for the first jobs to run on each storage server when the storage cache is *cold* (C) and for the subsequent jobs which run when the cache is *warm* (W). From these graphs, we can draw several conclusions. First, BAD-FS equals or exceeds the performance of remote I/O or standalone caching for all of the workloads in all of the configurations. These workloads, described previously in this dissertation in Chapter 2, all have large degrees of either batch or pipeline data sharing. Note that workloads whose I/O consists entirely of endpoint data would gain no benefit from our system.

Second, the benefit of caching, either cooperatively or in standalone mode, is greater for batch-intensive workloads, such as BLAST, than it is for more pipe-intensive ones such as HF. In these pipe-intensive workloads, the important optimization is pipeline localization, which is performed by both BAD-FS and standalone.

Third, cooperative caching in BAD-FS can outperform standalone both during cold and warm phases of execution. If the entire batch data set fits on each storage server, then cooperative caching is only an improvement while the data is being initially paged in. However, should the data exceed the capacity of any of the caches, then cooperative caching, unlike standalone, is able to aggregate the cache space and fit the working set.

This benefit of cooperative caching with warm caches is illustrated in the BLAST measurements in the graph on the left of Figure 3.7. Logfile analysis showed that two of the storage servers had slightly less cache space ($\approx$500 MB) than was needed for the total BLAST batch data ($\approx$600 MB). As subsequent jobs accessed these servers, they were forced to refetch data. Refetching it from the wide-area home server in the standalone case was much more expensive than refetching from the cooperative cache as in BAD-FS. With a local-area home server this performance advantage disappears. The different behavior of these two servers also explains the increased variability shown in these measurements.

Fourth, the penalty for performing remote I/O to the home node is less severe but still significant when the home node is in the same local-area network as the execute cluster. This result illustrates that BAD-FS can improve performance even when the bandwidth to the home server is not obviously a limiting resource.

Finally, comparing across graphs we make the further observation that BAD-FS performance is almost independent of the connection to the home server when caches are cold and becomes independent once they are warm. Using I/O scoping, BAD-FS is able to achieve local performance in remote environments.

3.3.8 In the Wild

Thus far, our evaluations have been conducted in controlled environments. We conclude our experimental presentation with a demonstration that BAD-FS is capable of operating in an uncontrolled, real world environment.

We created a wide-area BAD-FS system out of two existing batch systems. At the University of Wisconsin (UW), a large Condor system of over one thousand CPUs, including workstations, clusters, and classroom machines, is shared among a large number of users. At the University of New Mexico (UNM), a PBS system manages a cluster of over 200 dedicated machines.

We established a personal scheduler, catalog, and home storage server for our use at Wisconsin and then

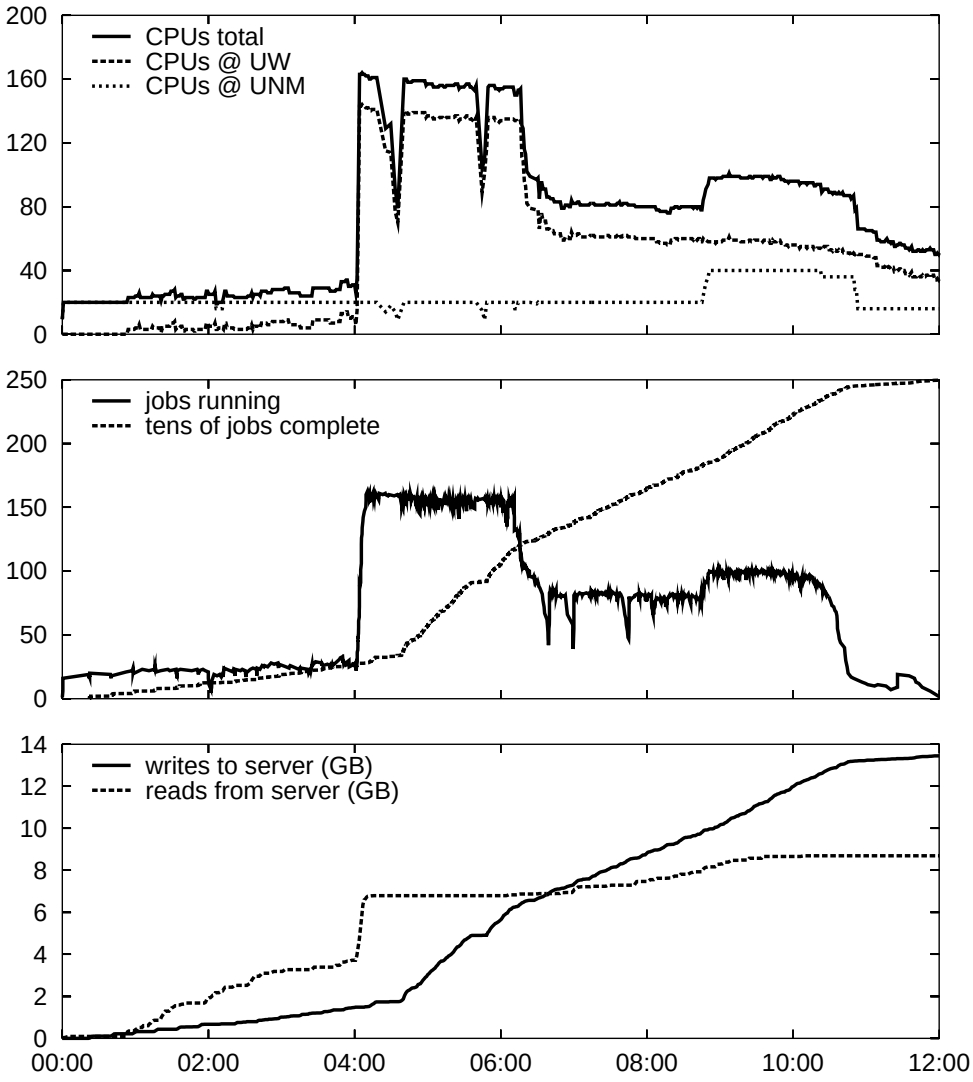


Figure 3.8: **In the Wild** *These three graphs present a timeline of the behavior of a large CMS workload run using BAD-FS. The workload consisted of 2500 CMS pipelines and was run wherever resources could be scavenged from a collection of CPUs at the University of New Mexico running PBS and from CPUs at the University of Wisconsin running Condor. The topmost timeline presents the total number of CPUs, the middle shows number of jobs running and cumulative jobs completed, and the bottom shows the cumulative traffic incurred at the home storage server.*

submitted a large number of BAD-FS bootstrap jobs to both batch systems without installing any special software at either of the locations. We then directed the scheduler to execute a large workload consisting of 2500 CMS pipelines using whatever resources became available.

Figure 3.8 is a timeline of the execution of this workload. As expected, the number of CPUs available to us varied widely, due to competition with other users, the availability of idle workstations (at UW), and the vagaries of each batch scheduler. UNM consistently provided twenty CPUs, later jumping to forty after nine hours. Two spikes in the available CPUs between 4 and 6 hours are due to the crash and recovery of the catalog server; this resulted in a loss of monitoring data, but not of running jobs.

The benefits of cooperative caching are underscored in such a dynamic environment. In the bottom graph, the cumulative read traffic from the home node is shown to have several hills and plateaus. The hills correspond to large spikes in the number of available CPUs.

Whenever CPUs from a new subnet begin executing, they fetch the batch data from the home node. However, smaller hills in the number of available CPUs do not have an effect on the amount of home read traffic because a new server entering an already established cooperative cache is able to fetch most of the batch data from its peers.

Note further that BAD-FS is able to complete the workload in a relatively short amount of time compared to remote I/O. Using the measurements for CMS from Figure 3.7, we can compare the BAD-FS runtime to the expected runtime we would achieve if we ran the same workload using remote I/O over a wide-area network. Even making generous assumptions such as an infinite ability to scale at the home node and an always available pool of the maximum number of CPUs that we saw in the BAD-FS run, using remote I/O would still take over 29 hours compared to just 12 using BAD-FS.

Finally, Figure 3.8 illustrates that both the design and implementation of BAD-FS are suitable for running I/O intensive, batch-pipeline workloads across multiple, uncontrolled real world clusters. Through failures and disconnections, BAD-FS continues making steady progress, removing the burden from the user of scheduling, monitoring, and resubmitting jobs.

3.4 Discussion

Allowing external control has long been recognized as a powerful technique to improve many aspects of system performance. By moving control to the external user of a system, that system allows the user to dictate the policy that is most appropriate to the individual nature of their work. Systems lacking mechanisms for external control can only speculate. However, many systems have proven to be adept at speculation and work well for the majority of their workloads. In this paper we have argued that the distinct nature of batch-pipeline workloads is not well matched by the design of traditional distributed file systems and the need therefore for external control is greater.

We have described BAD-FS, a distributed file system that exposes internal control decisions to an external scheduler. Using basic knowledge of workload characteristics, the scheduler carefully manages remote resources and facilitates the execution of I/O intensive batch jobs on both wide-area and local-area clusters. Through synthetic and real workload measurements in both controlled and uncontrolled environments, we have demonstrated the ability of BAD-FS to use workload specific knowledge to improve throughput by selecting appropriate storage policies in regards to I/O scoping, space allocation and cost-benefit replication.

However, although it does include some rudimentary scheduling policies, the main focus of this chapter is not a detailed study of data-driven scheduling for batch-pipeline workloads. Rather the focus as we have presented it

here is on the design of the distributed file system that allows data-driven scheduling.

We now turn in Chapter 4 to a more detailed focus on data-driven scheduling. Specifically, we answer the question of how a batch scheduler can leverage external storage control to produce a data-driven schedule for executing data-intensive batch-pipeline workloads in environments in which storage is scarce.

Chapter 4

Scheduling Batch-Pipeline Workloads

In Chapter 3 we show how a batch-aware distributed file system can allow *data-driven scheduling* of batch-pipeline workloads by transferring control of storage decisions from the storage layer to the batch scheduling system. Data-driven scheduling is needed whenever space is constrained, *i.e.* whenever the total amount of data accessed by a batch-pipeline workload exceeds the storage capacity of the compute resources.

Although current CPU-centric scheduling policies can be used to schedule space-constrained batch-pipeline workloads without introducing any correctness violations, they can incur orders of magnitude throughput loss as we demonstrate in Chapter 3. Recent trends in increasing dataset sizes in batch computing [43, 53] and the observation that this growth is outpacing the ability of computers to process data [53] lead to an ever-increasing need to supplement these current CPU-centric scheduling policies with data-driven policies as well.

While we do present some rudimentary scheduling policies in Chapter 3, the focus there is on the design of the distributed file system that allows data-driven scheduling. Now, in this chapter, we examine in greater and more formal detail how data-driven scheduling is implemented through the data allocation and job placement decisions made by the data-aware batch scheduler. To do so, we introduce and validate our simulator, BAD-Sim, that allows for a shorter development cycle as well as a clearer view into the intricacies of the distributed system.

We then codify some simplifying assumptions about the workloads into a new abstraction we call a canonical batch-pipeline workload. Using three representative canonical workloads and defining five possible scheduling strategies, we examine the effect of changing several different workload and environmental characteristics. We show how each of the different scheduling strategies is affected by each of these changes by analyzing several performance metrics such as CPU utilization, wide-area network traffic, and the total completion time of the workload.

Finally, we formalize predictive analytical models for each of the five possible scheduling strategies and demonstrate high levels of accuracy in their predictive abilities. We then show that across the entire set of experimentation that the models usually predict the "best" strategy and more importantly that they *never* predict the "worst."

4.1 Methodology

To more closely examine scheduling decisions within the BAD-FS framework, we developed a detailed discrete event batch computation simulator, BAD-Sim. This simulator consists of three main parts: one, the base compute

platform, two, the BAD-FS distributed file system and three, the scheduler which dictates job and data allocations.

The base compute platform is a simulated set of interconnected computers. Each simulated computer has a disk, a buffer cache, and a network. These components are modeled as queues with capacities, bandwidths, and latencies. As we are interested in studying the scheduling of I/O intensive workloads in a grid environment, we did not attempt to simulate any "lower" level machine components such as buses or processors.

Built on top of the base compute platform, we further simulate the full BAD-FS distributed file system. Each simulated machine within the compute cluster therefore binds into a cooperative cache with its peers as well as exports the storage allocation mechanisms which allow higher level planning by the BAD-FS scheduler.

The third part of the simulator is the data-aware batch scheduler which gathers both workload and environmental information and then creates a capacity-aware plan for workload execution. The scheduler uses the storage allocation mechanisms exported from the file system to prevent overallocations of storage by carefully coordinating data and job allocations.

The scheduler gets environmental and workload information from descriptive information provided to the simulator. The environmental description defines the various bandwidths, latencies, and capacities of both the cluster compute nodes as well as the home storage node.

Although BAD-Sim is capable of modeling multiple distinct compute clusters, in our study we examined only the base case of a single compute cluster linked via a wide-area network to a remote home storage server. The workload description file is identical to that presented in Chapter 3, with the only exception being that the jobs specified are not executables suited to running in a real system but rather point to files containing a descriptive trace of the job behavior. We include a validation of our simulator, including comparisons to our BAD-FS implementation, in the Appendix.

4.2 Defining Scheduling Allocations

We now focus on the scheduler's ability to formulate an allocation and job placement plan. Due to the large number of variables involved in this problem, we focus on the base case of executing a single workload on a single compute cluster and make three additional simplifying assumptions. First, we assume that the scheduler has access to detailed and accurate information both about the jobs within the workload and about the compute infrastructure. Second, we assume that the workloads are mostly uniform in structure; we refer to such uniform batch-pipeline workloads as *canonical*. Finally, we assume that the compute clusters consist of homogeneous machines. While we believe that these assumptions are reasonable, we will discuss relaxing them in Chapter 6.

4.2.1 Necessary Scheduling Information

Within the context of the simulation framework, we provided detailed workload information to the scheduler within the workload description file. As discussed in Chapter 3, we believe it is reasonable to expect the user to provide this information, although we will discuss relaxing this assumption later in this dissertation in Chapter 6.

The environmental information is also provided to the simulator as a seperate environmental description file. The information contained within this file is in regards to capacities, latencies, bandwidths, and failure rates. We believe it is reasonable to assume that compute nodes can accurately report their capacities and that accurate estimates for latencies, bandwidths, and failure rates can be obtained using a tool such as the Network Weather Service [115].

4.2.2 Canonical Workloads

To reduce the number of variables involved in developing a data-driven batch scheduler, we introduce the concept of canonical batch-pipeline workloads. Generally speaking, a canonical workload is a batch-pipeline workload in which all jobs have similar runtimes, all batch volumes are of the same size, all endpoint and pipelines volumes are of the same size, each pipeline is a single straight line (*i.e.* no multiple job dependencies), jobs read from a single batch volume which is read by all jobs at the same depth and each job reads from exactly one endpoint or pipeline volume and writes to one volume that is also either endpoint or pipeline.

For the sake of exposition, we introduce a new term, *private volumes*. Previously, when discussing batch-pipeline workloads, we referred to three different volume types, batch, endpoint and pipeline. Notice that both endpoint and pipeline volumes are accessed by only a single pipeline and that batch volumes are shared across many. We therefore combine endpoint and pipeline volumes into a single abstraction which we refer to as private. The value of combining these data types is to reduce the number of variables needed to describe the workloads. One trade-off is that it becomes harder to differentiate between endpoint-intensive workloads and pipeline-intensive workloads.

Specifically, a canonical workload is defined such that:

- All pipelines are of the same depth
- All jobs at a particular depth read from the same batch volume
- No batch volume is read by jobs at different depths
- Each job has either zero or one parents and either zero or one children
- Each job reads from exactly one private volume
- Each job writes to exactly one private volume
- Each job accesses the entire amount of data within each accessed volume
- The compute times for the jobs vary within a normal distribution

We show a canonical workload in Figure 4.1. Canonical workloads simplify data-driven scheduling because they can be represented using only six variables as shown as the top set of entries in Table 4.1. W_{Width} and W_{Depth} define the width and the depth of the workload. W_{Batch} is the size of each batch volume and $W_{Private}$ is the size of each private volume. Finally, W_{Run} and W_{Var} define the runtimes of the jobs within the workload; W_{Run} is the mean runtime of each job and W_{Var} is the expected variance such that the runtimes fit a normal distribution.

4.2.3 Compute Environment

Because we make the further simplifying assumption that the compute machines are homogeneous and we do not use information about disks and memory buffer caches to inform the scheduling decision, we can represent the compute cluster using only five variables as shown as the second set of entries in Table 4.1. Additionally, we present an illustration of our target compute environment labeled with these five variables in Figure 4.2.

Specifically, this environmental information can be reduced to five variables as listed in the first five rows of Table 4.1. The number of compute nodes in the compute cluster is C_{CPU}, the total amount of storage within the cluster (*i.e.* the sum of the storage from each node) is $C_{Storage}$, the rate of failure is $C_{Failure}$, the bandwidth between the cluster and the home storage site is C_{Remote}, and finally the bandwidth within the cluster is C_{Local}.

Although the base compute platform of the simulator does model the disks and the memory buffer caches of the compute nodes (as well as the home storage server), values for their capacities, latencies and bandwidths are not used by the scheduler and are therefore not included here. By not considering these values, the predictive scheduling model that we describe later in this chapter is greatly simplified without adversely affecting its predictive accuracy.

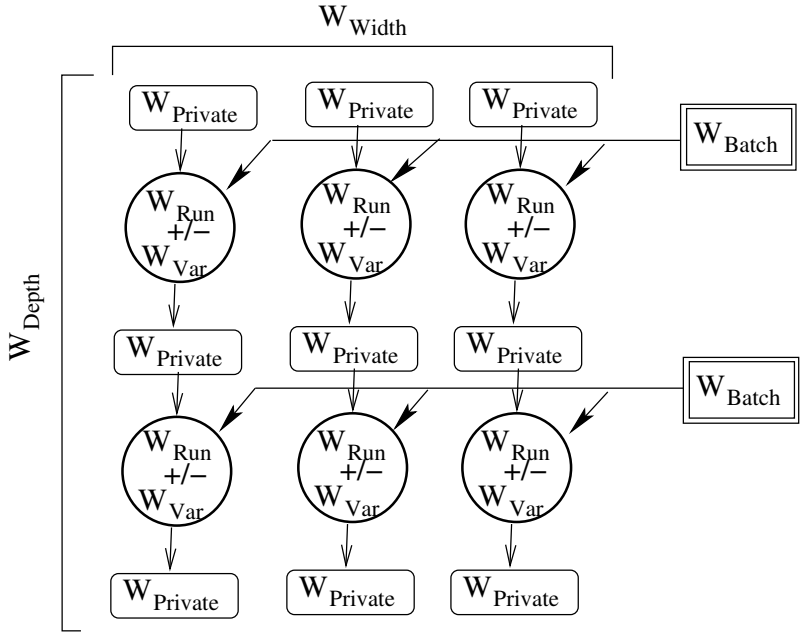


Figure 4.1: **A Canonical Batch-Pipeline Workload.** *We classify a batch-pipeline workload to be canonical when it is mostly uniform and show an example of one such canonical workload here. Circles represent jobs in the workload, double-edged rectangles represent batch data volumes, and single-edged rectangles are private data volumes. An advantage of canonical workloads is that they can be completely described using only six variables. W_{Width} and W_{Depth} define the width and the depth of the workload. W_{Batch} is the size of each batch volume and $W_{Private}$ is the size of each private volume. Finally, W_{Run} and W_{Var} define the runtimes of the jobs within the workload; W_{Run} is the mean runtime of each job and W_{Var} is the expected variance such that the runtimes fit a normal distribution. Extractions of endpoint outputs are not shown for the sake of exposition.*

4.2.4 Derived Values

Thus all information used by the scheduler to inform its allocation plan is encapsulated within a total of eleven variables: six to describe the canonical batch-pipeline and five for the compute environment. From these variables we derive additional values for use in our scheduling decision.

Several of these are constant and are shown in the third set of entries in Table 4.1. $D_{NumJobs}$ is the total number of jobs in a workload and is equal to the product of the depth and the width of the workload (*i.e.* $W_{Depth} \cdot W_{Width}$). $D_{TotBatch}$ is the total amount of batch data in a workload; as each depth has a distinct batch volume of size W_{Batch}, this value is equal to the product of the depth and this size (*i.e.* $W_{Depth} \cdot W_{Batch}$). Finally, $D_{TotPrivate}$ is the total amount of private data in a workload and is computed mathematically as $(W_{Depth}{+}1) \cdot W_{Private} \cdot W_{Width}$, which is the product of the amount of private data in a single pipeline ($(W_{Depth}{+}1) \cdot W_{Private}$) times the number of pipelines (W_{Width}). One thing that may be surprising about this equation is the $(W_{Depth}{+}1)$. This is because each pipeline of depth W_{Depth} has $(W_{Depth}{+}1)$ private volumes; this is obvious when considering that a pipeline of depth one actually has two private volumes.

Other derived values are variable dependending on the actual schedule produced and are shown as the fourth (and final) set of entries in Table 4.1, these are the number of batch volumes which can be allocated concurrently, V_{Batch}, the expected number of pipelines which can execute concurrently, V_{Exec}, and the number of phases in the workflow in which the batch data has been already cached, V_{Warm}. As these values are dependent upon the actual schedule produced, we will postpone their further explanation until each is encountered in the descriptions of the

W_{Width}	Workload width (*i.e.* number of pipelines)
W_{Depth}	Workload depth (*i.e.* number of jobs within each pipeline)
W_{Batch}	Size of each batch volume within a workload
$W_{Private}$	Size of each private volume within a workload
W_{Run}	Compute time of each job within a workload
W_{Var}	Compute time variability within a workload
C_{CPU}	Number of compute nodes in the compute cluster
$C_{Storage}$	Amount of storage available in the compute cluster
$C_{Failure}$	Rate of failure in the compute cluster
C_{Remote}	Remote bandwidth to the storage server
C_{Local}	Local bandwidth within the compute cluster
$D_{NumJobs}$	Derived total number of jobs $[W_{Depth} \cdot W_{Width}]$
$D_{TotBatch}$	Derived total amount of batch data $[W_{Depth} \cdot W_{Batch}]$
$D_{TotPrivate}$	Derived total amount of private data $[(W_{Depth}+1) \cdot W_{Private} \cdot W_{Width}]$
V_{Batch}	Number of concurrently allocable batch volumes
V_{Exec}	Number of executable pipelines in the steady state
V_{Warm}	Number of phases in which batch data is already cached

Table 4.1: **Glossary of Canonical Batch-Pipeline Scheduling Terminology.** *This table lists the variables used to schedule canonical batch-pipeline workloads. W values denote constant characteristics of the workload. C values denote constant characteristics of the compute environment. D values denote constants derived from the other values and V values are also derived but are variable depending on the actual schedule produced.*

different possible scheduling allocations.

4.2.5 Scheduling Objective

Using these six variables describing the workload and the five variables describing the environment, the batch scheduler can choose between different possible data allocations, each of which may lead to a different traversal order through the workload with various throughput implications. With an objective of maximizing the throughput of the workload (*i.e.* minimizing the total time to completion), the scheduler tries to avoid two potential pitfalls which may be possible in the different allocations and can reduce throughput. The first potential pitfall we have identified is an underutilization of the available CPUs. The second is sending redundant data over the wide-area network.

An underutilization of the available CPUs can happen in two different situations. The first occurs when the data required to run a single pipeline is greater than the storage available on a single compute node, such that it is necessary to impose *concurrency limits* on the workload thereby resulting in an underutilization of the available CPUs. In such a case, each pipeline, which computes on a single node, will use storage from multiple nodes. CPU underutilization can also be caused by the imposition of *barriers* within the workload. In a situation in which the maximum number of batch volumes which can be concurrently allocated is fewer than the total number of batch volumes, the scheduler can use barriers to ensure that only a subset of the batche volumes are accessed at any time. If the individual pipelines exhibit asynchrony in their runtimes, these barriers will result in an underutilization of the CPUs.

The second pitfall that is possible is the need to remove and then subsequently *refetch* batch volumes during the execution of a workload such that the wide-area network is redundantly utilized.

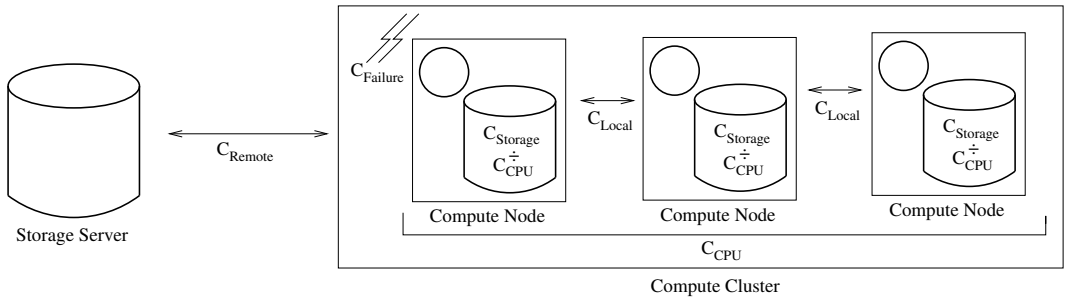


Figure 4.2: **The Target Compute Environment.** *This figure shows the target compute environment for running canonical batch-pipeline workloads in a distributed batch system. We show here the variables defining the various aspects of the environment that the scheduler uses to inform its scheduling plan. The number of compute nodes in the compute cluster is C_{CPU}, the total amount of storage within the cluster (i.e. the sum of the storage from each node) is $C_{Storage}$, the rate of failure is $C_{Failure}$, the bandwidth between the cluster and the home storage site is C_{Remote}, and finally the bandwidth within the cluster is C_{Local}.*

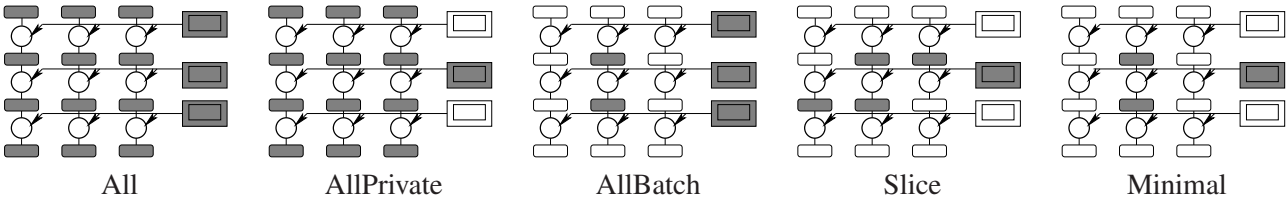


Figure 4.3: **Minimum Allocated Volumes for each Allocation Strategy** *These graphs show the minimum number of volumes which must be simultaneously allocated in order to make each allocation strategy possible. The double edged rectangles are batch volumes; the single are private. Jobs are represented as circles and the extractions are not shown for the sake of exposition. Whether a volume is allocated is indicated by shading it. For example, all volumes must be allocated for the All allocation as shown on the far left and only a single batch volume and two private volumes must be allocated to allow the Minimal allocation as shown on the far right.*

4.2.6 Defining Possible Scheduling Allocations

As shown Figure 4.3, we have identified five possible data allocations which influence the execution path and the performance of the workloads. Each of these allocations may suffer one or more possible throughput pitfalls. We now consider each of these five allocations in more detail.

The All allocation

In the unconstrained case, *All*, shown as the far right graph in Figure 4.3, every volume in the workload (*i.e.* all private and batch) fits within the total available storage. In such a scenario, the planning is straightforward as no possible schedule can result in adverse effects. Formally, All is possible in a canonical batch-pipeline workload whenever the total of all batch data ($D_{TotBatch}$) plus the total of all private data ($D_{TotPrivate}$) fits within the total amount of cluster storage ($C_{Storage}$), which translates mathematically to

$$D_{TotBatch} + D_{TotPrivate} \leq C_{Storage}. \tag{4.1}$$

A workflow traversal is shown in Figure 4.4 using the All allocation for a workload of width and depth three. Jobs are in one of three states: one, pending in which case they are shown as unfilled circles, two, executing

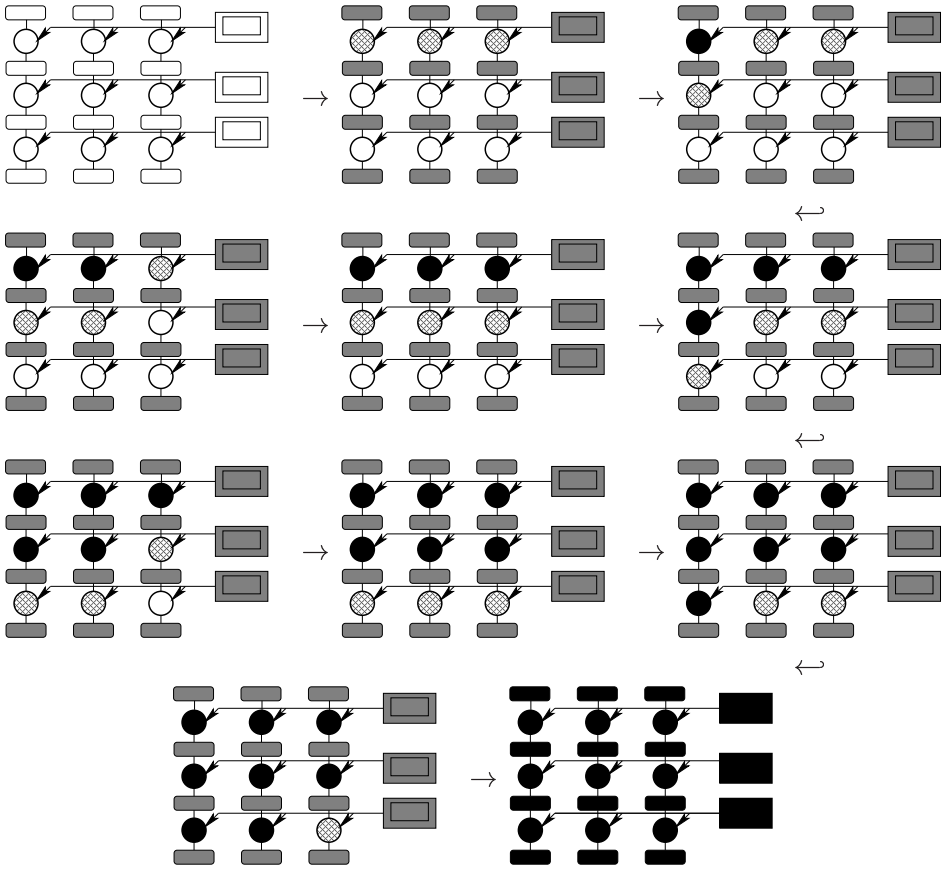


Figure 4.4: **Workflow Traversal for All.** *These graphs illustrate the workflow traversal for the All allocation which allows all batch and private volumes to be simultaneously allocated. These graphs, and subsequents one like them, are intended to be read in the standard English fashion from left to right, top to bottom (i.e. follow the arrows). Pending jobs and pending volumes are unfilled, executing jobs are shown with a cross-grid pattern, allocated volumes are shaded, and finished jobs and finished volumes are filled in black. In this case in which all volumes can be simultaneously allocated, the schedule is completely without data constraint and each job can begin executing as soon as its parent finishes.*

represented with a cross-grid pattern, and three, completed which are shaded black. Volumes are similar and have pending and completed states which are shown respectively as unfilled and black in the same manner as for jobs. However, volumes do not have an executing state as do jobs but rather are allocated between their pending and completed states; allocated volumes are indicated here by shading them.

This traversal assumes that jobs are synchronized such that jobs which begin executing at approximately the same time complete at approximately the same time. Further assumed is that there are at least three compute nodes so that each pipeline is able to execute concurrently. Given these two assumptions, the traversal will proceed such that the executing pipelines remain approximately, but not perfectly, in synchrony.

The traversal begins with the initial state in which no jobs are executing and no volumes are allocated as shown in the upper right graph. In the next graph, the scheduler has allocated all volumes and has begun executing the first job in each of the three pipelines. As the first job finishes in the top right graph, its child can immediately begin executing because all volumes needed for that job to execute have already been allocated. This continues until the bottom right graph in which the final job finishes and the volumes can be de-allocated.

Notice that because all volumes can be concurrently allocated that there are no limits on concurrency (*i.e.* there are always three jobs executing until the pipelines begin finishing at the end), there are no barriers imposed, and that there is no refetching of batch volumes. This is true for only the All allocation; as we examine more constrained allocations, we will begin to see barriers, concurrency limitations, and refetching of batch data.

The AllPrivate allocation

A second allocation, *AllPrivate*, is possible when all the private volumes and at least one batch volume fit within available storage,

$$W_{Batch} + D_{TotPrivate} \leq C_{Storage}. \tag{4.2}$$

This is the same as Equation 4.1 for the All allocation except that only one batch volume (W_{Batch}) need be allocated instead of all of them ($D_{TotBatch}$).

This minimum allocation for AllPrivate is shown in the graph second from the left in Figure 4.3 in which all private volumes and a single batch volume are allocated (as shown by being shaded). Notice again the similarty to the All allocation with the sole difference that only a single batch volume is allocated at any one time.

We know examine a workflow traversal for AllPrivate for the maximally constrained case in which only a single batch volume can be allocated at any one time. The number of batch volumes which can be concurrently allocated is one of our derived variables, V_{Batch}. Therefore, this maximally constrained case occurs whenever $V_{Batch} = 1$.

This constraint will exist whenever the sum of all private data ($D_{TotPrivate}$) plus the sum of the data in *two* batch volumes exceeds the total available storage but the sum of all private data plus only the data in *one* batch volume does fit within total available storage. For example, the maximum value of V_{Batch} for any workload in which each batch volume requires more than 50% of the available storage is 1. If the sum of the private volumes requires less than the remaining space after allocating one of these large volumes, then the AllPrivate allocation is possible with this constraint of $V_{Batch} = 1$.

Such a constrained traversal is shown in Figure 4.5; this traversal is similar to that in Figure 4.4 for All except that barriers are needed to ensure that only one batch volume is accessed at any one time. This difference is seen in the top right graph which shows the state of the workload after the first job completes. In this case, because the

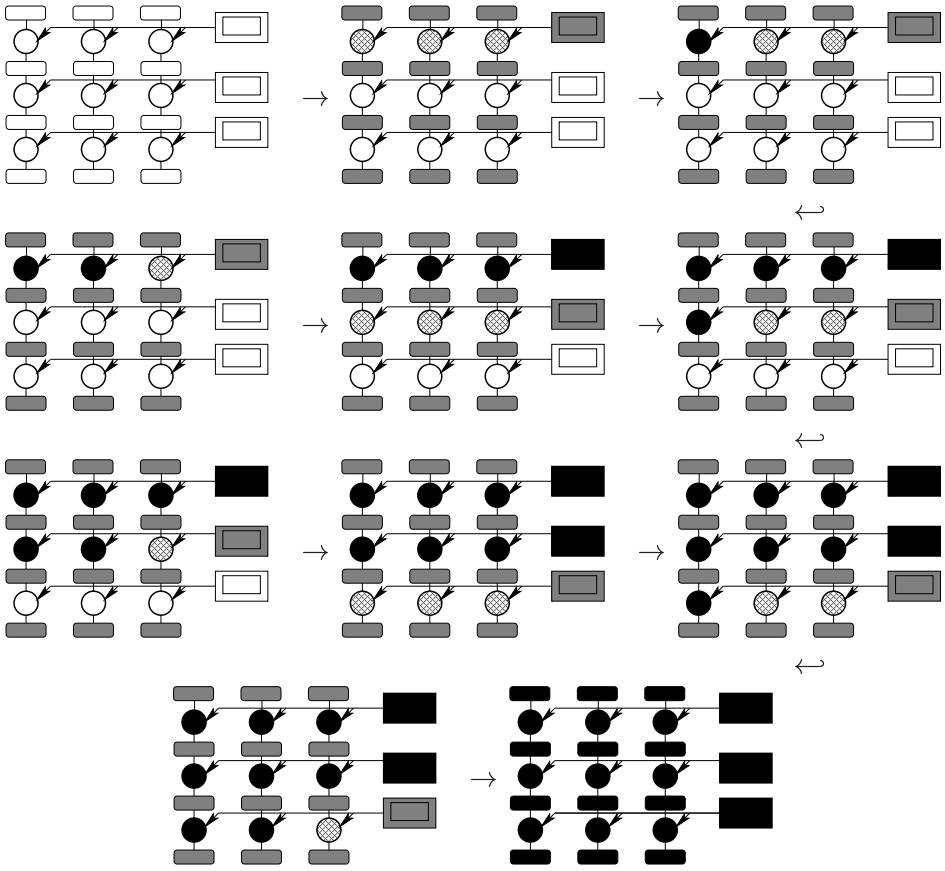


Figure 4.5: **Workflow Traversal for AllPrivate with $V_{Batch} = 1$.** *These graphs show the workflow traversal for the AllPrivate allocation in the maximally constrained case in which only one batch volume can be concurrently allocated (i.e. $V_{Batch} = 1$). In such a case, as is shown in the top right graph, barriers must be imposed such that jobs whose job dependencies are satisfied (in this case, the second job in the left-most pipeline), must wait for their sibling jobs to complete. Upon the completion of all jobs at a particular depth as is shown in the transistion between the fourth and fifth graphs here, the batch volume at that depth can be removed, thereby freeing storage to allow the next batch volume to be allocated and then the jobs that need that batch volume can begin executing. The duration of time that the CPUs here will be underutilized due to this barrier depends on the synchrony of the executing jobs (i.e.workload variability, W_{Var}).*

second batch volume cannot be simultaneously allocated along with the first, the second job in the first pipeline cannot begin executing. Even though its job dependencies are satisfied (*i.e.* its parent has completed), it cannot begin executing because its required data cannot be currently allocated. In contrast, in a All allocation as was shown in Figure 4.4, all jobs can begin executing as soon as their job dependencies are satisfied. In this case, however, a complete barrier is necessary such that no job can begin executing until *all* jobs at the previous depth have completed. Once all jobs at the previous depth have completed, the batch volume at that depth can be de-allocated, thereby freeing room for the next batch volume and allowing jobs at that next level to begin executing. The duration of these barriers depends on the degree of synchrony of the executing pipelines and will cause some underutilization of computation whenever the runtimes are not perfectly synchronized.

Notice that, although there may be CPU underutilization due to barriers, that there is no possibility of CPU underutilization due to concurrency limits since all of the private volumes can be simultaneously allocated. For this same reason, no batch volumes will need to be refetched because all jobs that use a batch volume can be executed before the traversal need remove their batch volume.

The AllBatch allocation

Conversely, if all the batch volumes fit without sufficient remaining space for all of the private volumes, an *AllBatch* allocation is possible so long as at least one pipeline can have access to both of its private volumes and thereby can execute as shown in the third graph from the left in Figure 4.3. A single executing pipeline will require two private volumes (both input and output), so AllBatch is formally possible whenever the sum of all batch data plus the data needed for two private volumes fits within available storage which translates mathematically to

$$D_{TotBatch} + 2W_{Private} \leq C_{Storage}. \tag{4.3}$$

As all batch volumes can be simultaneously allocated, no barriers need be imposed, nor will any batch volumes need be refetched in an AllBatch allocation. However, an underutilization of the compute capacity of the storage nodes may be incurred if fewer pipelines can simultaneously execute than the number of compute nodes. To explain this we use another derived variable, V_{Exec}, which refers to the number of pipelines which can concurrently execute.

For both of the previously described allocations, All and AllPrivate, this value was never constrained as all private volumes could be concurrently allocated and thus all pipelines could concurrently execute. With an AllBatch allocation however, only a subset of the total private volumes can be allocated and therefore only a subset of the total number of pipelines can concurrently execute.

We will give the exact derivation of this value when we present the predictive runtime model for the AllBatch allocation; suffice it here to say that this value will be maximally constrained at 1 when only a single pipeline can allocate two private volumes from the total available storage remaining after allocating the total amount of batch data.

A workflow traversal for AllBatch is shown in Figure 4.6 for this maximally constrained case in which only a single pipeline can concurrently execute (*i.e.* $V_{Exec} = 1$). This traversal shows the loss in utilization as only a single job can execute at any given time. Each pipeline must execute in its entirety before another can begin as is shown here in the transistion between the fourth and fifth graphs. Notice the depth-first traversal that results from this constrained schedule.

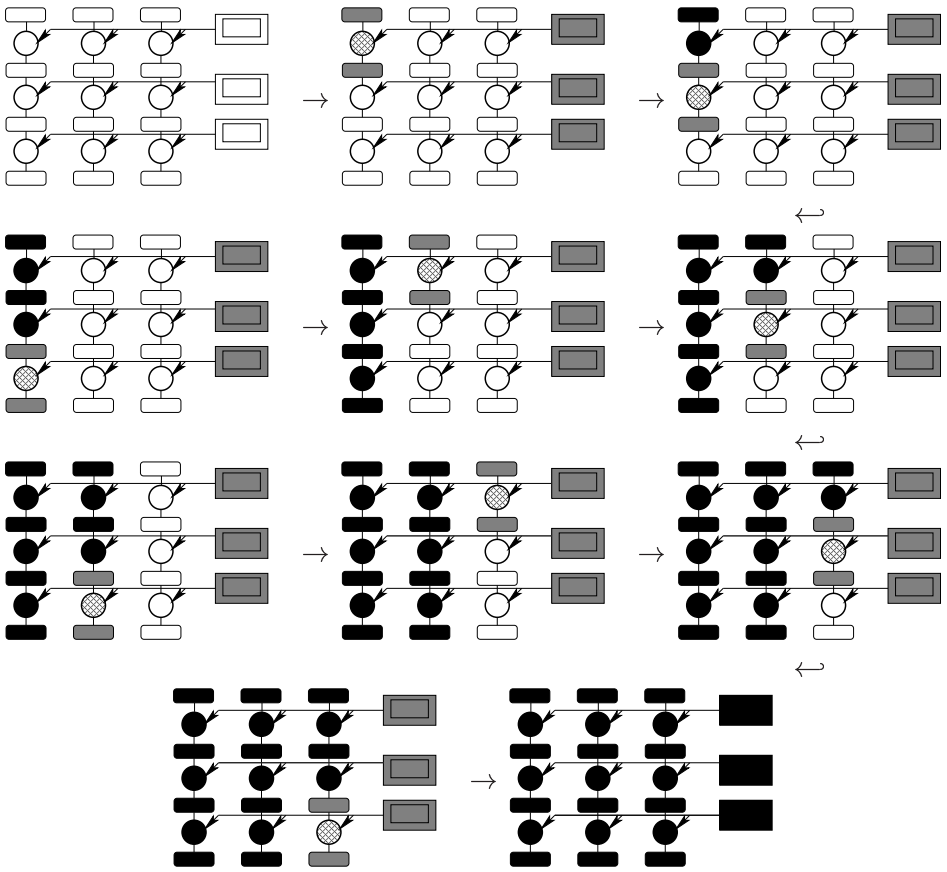


Figure 4.6: **Workflow Traversal for AllBatch with $V_{Exec} = 1$.** *These graphs show the workflow traversal for the AllBatch allocation in the maximally constrained case in which after allocating all batch volumes, there exists additional storage for only two private volumes. In such a case, the steady state concurrency of the workload is limited to just a single pipeline executing at a time (i.e. $V_{Exec} = 1$). This is shown in these graphs as only a single job can execute at any time and that once a pipeline begins, only jobs in that pipeline can executed until it completes. Once a pipeline is completed, another can begin as is shown in the transistion between the fourth and fifth graphs.*

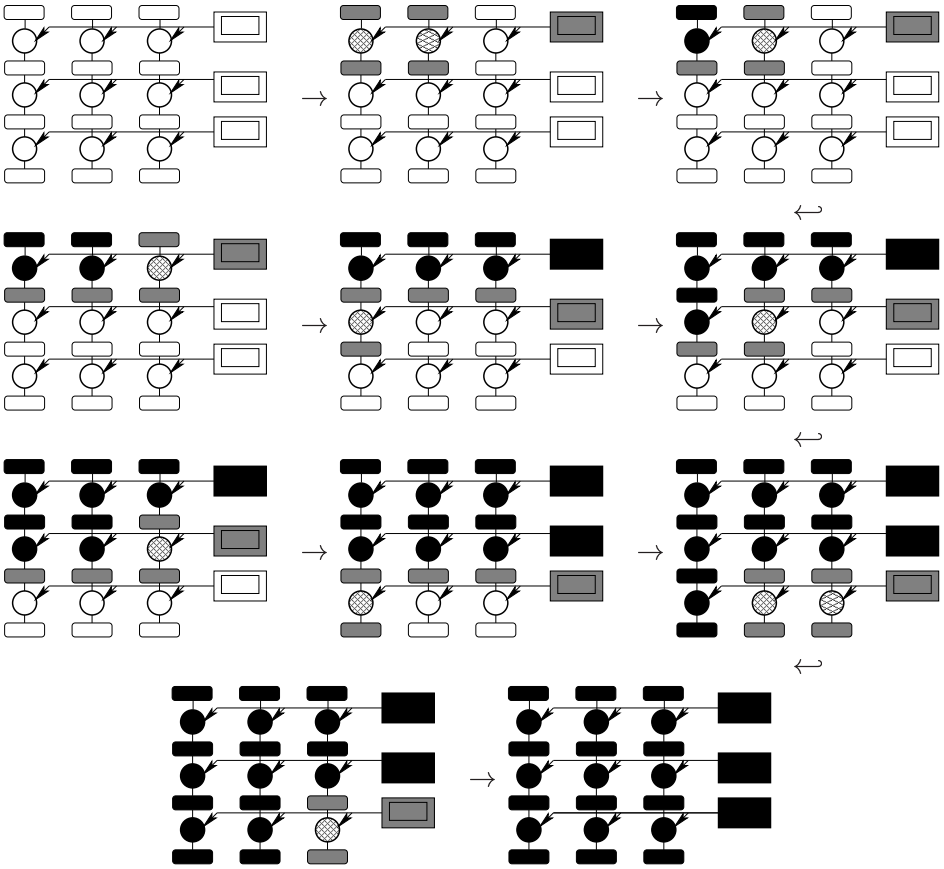


Figure 4.7: **Workflow Traversal for Slice with $V_{Exec} = 1$.** *These graphs show a workflow traversl for the Slice allocation in the maximally constrained case in which only a single pipeline can concurrently execute (i.e. $V_{Exec} = 1$) except during the very beginning and end of the workflow. As each job finishes and its input private volume can be released, an additional sibling can then execute. Only after all jobs at a depth have finished, can the batch volume at that depth be removed and the workflow can descend. This descent is shown here in the transistion between the fourth and fifth graphs when the final job at depth 1 finishes. Notice how this allocation results in a breadth-first traversal.*

Although CPU underutilization may occur due to an inability to allocate sufficient private volumes to run at maximum utilization, that CPU underutilization due to barriers is not possible in an AllBatch allocation as all of the batches are concurrently allocated. For this same reason, no batch volumes will need be refetched.

The Slice allocation

The fourth graph from the left in Figure 4.3 shows the minimum allocated volumes for the Slice allocation strategy which is possible whenever an entire horizontal *slice* of the workload executes to completion before any jobs at the next horizontal slice begin executing. A horizontal slice of the workload has a storage requirement of one batch volume and one private volume for each pipeline *plus* at least one more private volume so that at least one pipeline can have access to both of its private volumes and therefore be able to execute and ensure progress of the workload. More formally, Slice is possible whenever

$$W_{Batch} + W_{Private} \cdot W_{Width} + W_{Private} \leq C_{Storage}. \tag{4.4}$$

Because the entire horizontal slice will execute before moving the workload deeper, no batch refetch is neces-

sary in a Slice allocation. However, barriers are possible as they were in an AllPrivate allocation.

Also, as was the case for AllBatch but was not the case for AllPrivate, an underutilization of computation may occur when the remaining storage after allocating an entire horizontal slice allows only a subset of the pipelines to allocate a second private volume and therefore be able to execute.

We will provide the exact derivation of the maximum concurrency, V_{Exec}, when we present the predictive runtime model for the Slice allocation; suffice it here to say that this value will be maximally constrained at 1 when only a single pipeline can allocate its second private volume from the total available storage remaining after allocating the total horizontal slice. A workflow traversal for the Slice allocation is shown in Figure 4.7 for this maximally constrained case.

Notice that in the steady state once all pipelines have begun executing that the maximum number of executing jobs is the expected value of 1. This is not true for the entire duration of the workflow however, as two jobs are able to execute concurrently at the very beginning and end of the workload. As the workload "ramps up" an increasing number of private volumes are allocated. In this case, the maximum number of private volumes that can be allocated is four. In the steady state, after each of the three pipeline has begun, one private volume for each pipeline is allocated leaving space for only one additional private volume thereby allowing only a single job access to both its input and output private volumes and limiting concurrency to one.

However, during the ramp up phase, not every pipeline has begun and there is additional space. Since four private volumes can be allocated at any time, this allows two jobs to execute at the very beginning of the traversal. The same effect is seen at the end after the first pipeline finishes, no private volumes for it need remain allocated thereby freeing storage, allowing an additional private volume to be allocated and therefore an additional job to execute.

Because of this difference in concurrency between the ends of the workflow and the middle, the average concurrency, $V_{ExecAve}$, for the Slice allocation is not the same as the steady state concurrency, V_{Exec}. This discrepancy exists only in the Slice allocation, the value for $V_{ExecAve}$ and $V_{ExecAve}$ in the other allocations are the same as the concurrency for the other allocations is more consistent throughout the entire workflow. The exact derivation for both the $V_{ExecAve}$ and V_{Exec} values for the Slice allocation will be shown when we discuss its predictive runtime model.

Notice finally that although there may be CPU underutilization due to barriers (not shown in the traversal in Figure 4.7) and CPU underutilization due to concurrency limits (as shown), notice that the Slice allocation avoids refetching any batch data. As every private volume from a particular slice is able to be concurrently allocated, the workflow is able to completely exhaust all jobs at a particular depth before descending. This exhaustion of a depth allows the Slice allocation to avoid any batch volume refetch even when it is maximally constrained as shown here. Notice the breadth-first traversal that appears here as a result of this allocation.

The Minimal allocation

The most constrained possible allocation, as shown on the far right of Figure 4.3, is *Minimal*, in which only a *partial* horizontal slice of the workload can execute before storage is exhausted and the workload is forced to move deeper. A Minimal allocation is possible so long as at least one pipeline can have access to its batch volume and both of its private; more formally whenever

$$W_{Batch} + 2W_{Private} \leq C_{Storage}. \tag{4.5}$$

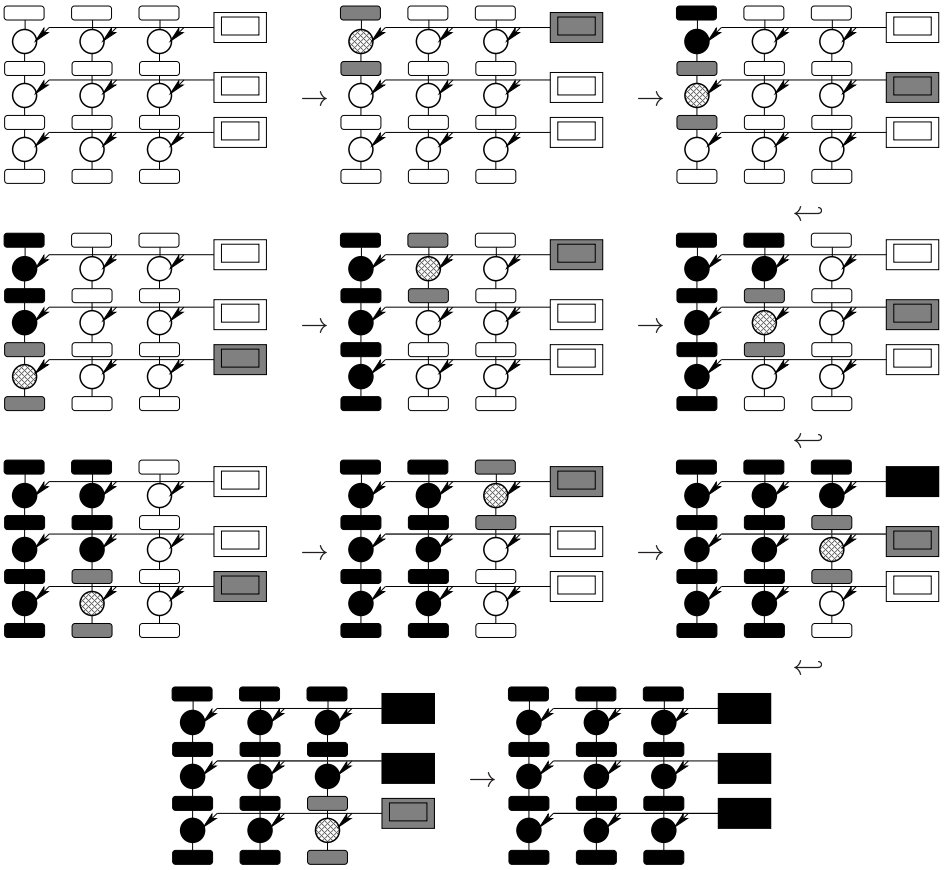


Figure 4.8: **Workflow Traversal for Minimal with $V_{Exec} = 1$.** *These graphs show a workflow traversal for the Minimal allocation in the maximally constrained case in which only a single pipeline can execute at any given time (i.e. $V_{Exec} = 1$). The traversal is identical to that shown for the AllBatch allocation in Figure 4.6 except that each batch volume is removed and then refetched for each pipeline.*

Minimal can suffer from all three pitfalls; it will definitely refetch batch volumes and might suffer from an under-utilization of computation due to either concurrency limitations or to barriers.

Each batch volume will be refetched at least once because each will be fetched for every partial horizontal slice of the workload which executes. More formally, the number of batch refetches is constrainted between 1 and $W_{Width} - 1$, depending on the ratio of V_{Exec} to W_{Width} (*i.e.* $\left\lceil \frac{W_{Width}}{V_{Exec}} \right\rceil - 1$).

The expected concurrency, V_{Exec}, in a Minimal allocation may also be constrained. By definition, whenever the Minimal allocation is possible and the Slice allocation is not, the maximum concurrency of the workload will necessarily be smaller than the width of the workload (*i.e.* $V_{Exec} < W_{Width}$). However, whether this concurrency limit actually results in an underutilization of CPU depends on the number of nodes in the compute cluster (C_{CPU}). The exact derivation of the V_{Exec} value will be provided in the predictive runtime model; suffice it here to say that this value will be maximally constrained at 1 whenever no additional volumes can be allocated after allocating a single batch volume and two private volumes (*i.e.* after allocating the minimum volumes required to execute a a single pipeline).

Figure 4.8 shows a workflow traversal for a Minimal allocation for the maximally constrained case in which $V_{Exec} = 1$. This traversal appears similar to that in Figure 4.6 for the AllBatch allocation except that each batch volume must be removed in order for the workflow to proceed to a lower depth. In such a maximally constrained case, as illustrated in this traversal, each batch volume must be fetched for every pipeline. Notice, as was the case for the AllBatch allocation, that the Minimal allocation with concurrency constraints results in a depth-first traversal of the workload.

The Remote allocation

Finally, there exist workloads whose combination of batch and private volume sizes is sufficiently large that none of the previous allocations are possible. We refer to this as *Remote*. Strictly speaking however some Remote workloads could still technically execute if they did not attempt to cache their batch data, but rather used remote I/O to fetch it as necessary. Although this is feasible, we do not consider it further in this work as we are more interested in the challenges when allocations are possible but constrained.

Further we note that this ability to use remote I/O to fetch batch data should be used judiciously and probably only at the discretion of the user. As was seen in Chapter 3, using remote I/O for batch-pipeline workloads results in a drastic performance "cliff." Such extreme performance differences should not be made transparent to the user. In such a case, it seems reasonable that the scheduler could cowardly refuse to use remote I/O for an Remote workload until giving the user an opportunity to reconfigure the workload or to explicitly request remote I/O. Finally although this technique of using remote I/O for the batch data does allow some Remote workloads to execute, note that there is no technique which can execute workloads that generate excessive private output data.

4.3 Possible Volume Sizes for the Scheduling Allocations

Using the formulae for determining when each allocation strategy is possible reveals that, depending upon the amount of storage a workload needs, multiple scheduling allocations may be possible. These formulae are summarized here in Table 4.2.

We then plot these formulae for different values of workload width, W_{Width}, and workload depth, W_{Depth}, and show these graphs in Figure 4.9. These graphs show in the areas under the lines where each strategy is possible for

Allocation	Minimum Storage Needed
All	$W_{Depth} \cdot W_{Batch} + (W_{Depth}+1) \cdot W_{Private} \cdot W_{Width}$
AllPrivate	$W_{Batch} + (W_{Depth}+1) \cdot W_{Private} \cdot W_{Width}$
AllBatch	$W_{Depth} \cdot W_{Batch} + 2W_{Private}$
Slice	$W_{Batch} + W_{Private} \cdot W_{Width} + W_{Private}$
Minimal	$W_{Batch} + 2W_{Private}$
Remote	$\emptyset$

Table 4.2: **Minimum Storage Needed.** *This table summarizes the minimum amount of storage needed for each of the scheduling allocations we have defined.*

all possible values of batch volume size, W_{Batch}, on the x-axis, and private volume size, $W_{Private}$, on the y-axis. These volume sizes are shown as a percentage of $C_{Storage}$. They are presented in tabular format to show the effect that increasing workload depths (W_{Depth}) and widths (W_{Width}) has on each strategy. There are several points to notice in these graphs.

The first thing to notice is that in all cases, the maximum batch volume approaches the full 100% of $C_{Storage}$ but that the maximum private volume is only 50% of $C_{Storage}$. This difference is due to the fact that, in order to execute, a job must have access to only a single batch volume but to two private ones.

Further, we see that each allocation strategy is effected differently by increasing values of workload depth and width. Comparing across the top row as the workload width increases we observe that only the possible areas for the AllBatch and Minimal allocations are robust to increasing workload widths. Conversely by looking at the column of left-most graphs, we see that only the possible areas for the Slice and Minimal allocations are robust to increasing the depth of the workload.

Looking in more detail at the effect of width and depth to each particular allocation, we observe that the All allocation is sensitive to both width and depth; as the width increases, the maximum size of batch volumes is unaffected but the maximum size of private volumes that can fit within a fixed $C_{Storage}$ decreases. This is intuitive because as the width increases, the total amount of batch data remains constant but the total amount of private data increases. Conversely, increasing the depth decreases both the maximum size of private volumes as well as the maximum size of batch volumes because an increase in depth results in a larger amount of both batch and private data.

The behavior for the AllPrivate allocation is similar to that of All with the exception that the maximum size of the batch volumes is constant in AllPrivate since it need only allocate a single volume at a time. The AllBatch allocation shows effectively the converse behavior to AllPrivate; it is sensitive to depth but not width and the maximum private volume size remains constant. Again this is intuitive because the AllBatch allocation requires simultaneously allocating all batch volumes but only two private volumes.

As expected, the Slice allocation which need allocate only a single horizontal slice of the workload is sensitive to width but robust to depth. Finally, the Minimal allocation, needing only a single batch and two private volumes, is constant in both directions. Results for these relationships between workload depth and width and the maximum volume sizes are summarized in Table 4.3.

These graphs allow comparison of the different possible areas for each of the five allocations which can aid in the decision making process by which the scheduler selects an allocation. Notice initially that only in the base case where W_{Width} and W_{Depth} are both 1 are the possible areas the same for each allocation. For each other possible combination of W_{Width} and W_{Depth}, the areas are different. For very large data sets in which the batch volume

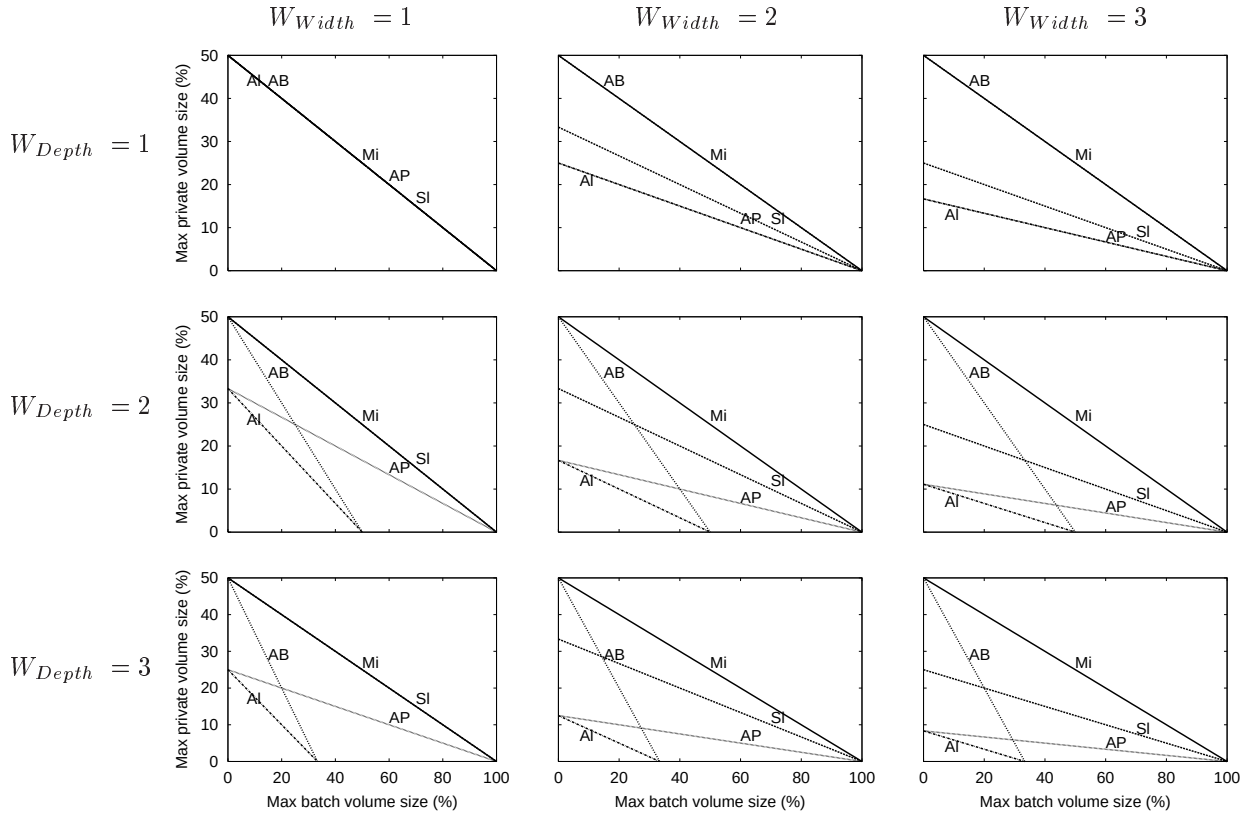


Figure 4.9: **Possible Volume Sizes for the Different Allocations.** *These graphs show as the area under the lines when each scheduling allocation is possible for varying batch and private volume sizes for canonical batch-pipeline workloads. The names of the allocation strategies have been abbreviated here: Al is All, AB is AllBatch, AP is AllPrivate, Sl is Sliceand Mi is Minimal. On the x-axis is the size of the batch volumes and the y-axis shows the size for private. Notice that for all cases, the maximum batch volume size can approach 100% of total available storage but that the maximum private volume size can only approach 50%. This is because an executing job needs access to only a single batch volume but to two private volumes.*

size exceeds the total cluster storage or where two private volumes do, there will be no allocations possible. For smaller volume sizes, in some cases there may only be one possible allocation. For example, for a width and depth both of 3, only the Minimal and AllBatch allocations are possible for private volume sizes approaching 50% of the total cluster storage. Conversely, at the same width and depth of 3 but with a batch volume size approaching 100%, only the AllPrivate, the Slice, and the Minimal allocations are possible. For both a large private size and a large batch size, only the Minimal allocation is possible (for example, when $W_{Depth} = 3, W_{Width} = 3, W_{Batch} = 49, W_{Private} = 24$). Finally, for small values for $W_{Private}$ and W_{Batch}, all allocations are possible.

The key observation here is that when multiple allocations are possible, the scheduler needs some additional criteria by which to choose a scheduling allocation.

4.4 Predictive Analytical Modelling

To determine which allocations are *preferable* when multiple allocations are possible, we develop an analytical model for predicting the runtimes for each of the scheduling allocations. This predictive model is relatively simple and was added to our simulated scheduler using fewer than 500 lines of code. The algorithms extend logically and build upon the equations previously defined in Table 4.2 and use only the base eleven variables from Table 4.1: the

	$W_{Width} \Uparrow$	$W_{Depth} \Uparrow$
All	$W_{Private} \Downarrow$	$W_{Batch} \Downarrow, W_{Private} \Downarrow$
AllPrivate	$W_{Private} \Downarrow$	$W_{Private} \Downarrow$
AllBatch	$\emptyset$	$W_{Batch} \Downarrow$
Slice	$W_{Private} \Downarrow$	$\emptyset$
Minimal	$\emptyset$	$\emptyset$

Table 4.3: **Effect of W_{Width} and W_{Depth} on Maximum Volume Sizes.** *This table shows the effect that increasing width, W_{Width}, and increasing depth, W_{Depth}, has on the maximum size of private and batch volumes within a canonical batch-pipeline workload for each of our five studied allocations. Consider, for example, the All allocation. An increase in the width of the workload reduces the maximum size of private volumes for a All allocation but has no effect of the maximum size of batch volumes; increasing the depth however decreases the maximum size of both private and batch volumes. Effects on the other allocations can be interpreted similarly.*

five environmental variables, C_{CPU}, the number of compute nodes, $C_{Storage}$, the total amount of available storage in the compute cluster, $C_{Failure}$, the failure rate, C_{Remote} the remote network bandwidth between the compute cluster and the home storage server, and C_{Local}, the local network bandwidth within the compute cluster, and the six workload variables, W_{Width} and W_{Depth}, the width and depth of the workload, W_{Batch} and $W_{Private}$, the size of each batch and private volume, and finally W_{Run} and W_{Var}, the compute times and their variability.

Notice that many of the low level characteristics, such as the disks and buffer caches, of the cluster environment are *not* considered in these predictive models. As we will see, this simplification may cause some absolute inaccuracies in the model's ability to predict absolute runtimes but does not adversely affect the model's ability to predict the relative performance of the different scheduling allocations. As this relative performance is what is used for predictive scheduling, this simplification and corresponding loss of absolute accuracy is justified.

To predict the total time to completion for these workloads, we compute how many phases will be required to run these workloads. The number of these phases is generally the total width of the workload divided by our anticipated degree of concurrency. Each phase then consists of executing some number of pipelines.

The runtime for each phase is computed to be the total amount of compute time within that phase plus the time needed to access the total amount of private and batch data for a pipeline within that phase. Notice that our model assumes that the runtimes for all pipelines within a phase can be computed as a single number. Also our model does not consider any costs due to contention that may be incurred depending on the degree of concurrency. As we will see in our evaluation, these simplifications cause the model to tend to underpredict the absolute runtimes but does not effect its relative predictions comparing across scheduling allocations.

4.4.1 Predicting Runtime for All

Specifically, for the All allocation, the total time to completion for a workload includes a "cold" phase ($T_{ColdPhase}$) during which the batch data is fetched from the remote home storage server followed by some number, V_{Warm}, of "warm" phases each of which takes time $T_{WarmPhase}$ during which the cached batch data is fetched locally within the compute cluster. This is formalized as

$$T_{Total} = T_{ColdPhase} + V_{Warm} \cdot T_{WarmPhase} \tag{4.6}$$

which also appears in Figure 4.10 which shows the complete predictive algorithm for the All the AllPrivate, and the AllBatch allocations.

$$
\begin{aligned}
T_{Total} &= T_{ColdPhase} + V_{Warm} \cdot T_{WarmPhase} \\
T_{ColdPhase} &= T_{ColdBatch} + T_{PrivateRead} + T_{PrivateWrite} + T_{Compute} \\
T_{ColdBatch} &= BW(C_{Remote}, C_{Local}) \cdot D_{TotBatch} \\
BW(C_{Remote}, C_{Local}) &= \frac{1}{\frac{1}{C_{Remote}} + \frac{1}{C_{Local}}} \\
D_{TotBatch} &= W_{Depth} \cdot W_{Batch} \\
T_{PrivateRead} &= BW(C_{Remote}, C_{Local}) \cdot W_{Private} + C_{Local} \cdot W_{Private} \cdot (W_{Depth} - 1) \\
T_{PrivateWrite} &= C_{Local} \cdot W_{Private} \cdot (W_{Depth} - 1) + BW(C_{Remote}, C_{Local}) \cdot W_{Private} \\
T_{Compute} &= W_{Run} \cdot W_{Depth} \\
V_{Warm} &= \left\lceil \frac{W_{Width}}{min(V_{Exec}, C_{CPU})} \right\rceil - 1 \\
V_{Exec} &= W_{Width} \\
T_{WarmPhase} &= T_{WarmBatch} + T_{PrivateRead} + T_{PrivateWrite} + T_{Compute} \\
T_{WarmBatch} &= C_{Local} \cdot D_{TotBatch}
\end{aligned}
$$

Figure 4.10: **The Predictive Model for the All Allocation.** *These equations form the predictive model which the scheduler uses to predict the total time to completion for workloads scheduled using the All allocation.*

Cold phase

The runtime for the cold phase consists of the sum of the time to fetch the batch data from the remote node ($T_{ColdBatch}$), the time to read the private data ($T_{PrivateRead}$), the time to write the private data ($T_{PrivateWrite}$), and the compute time ($T_{Compute}$):

$$T_{ColdPhase} = T_{ColdBatch} + T_{PrivateRead} + T_{PrivateWrite} + T_{Compute}. \tag{4.7}$$

Each of these times are estimated using some set of our eleven constants describing the workload and the compute environment.

The time to read the batch data remotely ($T_{ColdBatch}$) is the product of the serial bandwidth of sending data through the remote and local bandwidths and the total amount of batch data ($D_{TotBatch}$):

$$T_{ColdBatch} = BW(C_{Remote}, C_{Local}) \cdot D_{TotBatch} \tag{4.8}$$

The serial bandwidth, BW, for a set of components is the inverted sum of the inverted bandwidths for each component or mathematically,

$$\frac{1}{\sum_{bw=1}^{n} \frac{1}{bw}} \tag{4.9}$$

In this case, the serial bandwidth to read data remotely is

$$BW(C_{Remote}, C_{Local}) = \frac{1}{\frac{1}{C_{Remote}} + \frac{1}{C_{Local}}} \tag{4.10}$$

The derived value for the total amount of batch data is the product of the workload depth and the batch volume size:

$$D_{TotBatch} = W_{Depth} \cdot W_{Batch} \tag{4.11}$$

The private input data is read either from the home node for the first job in each pipeline or from the local cluster for subsequent jobs. The final private output is written first to the local cluster and then later extracted to the home storage server.

The time to read the private data, $T_{PrivateRead}$, is computed to be the time to read one private volume from the remote storage server at the serial bandwidth of C_{Remote} and C_{Local} plus the time to read the remaining $W_{Depth} - 1$ private volumes from the local compute cluster at C_{Local}:

$$T_{PrivateRead} = BW(C_{Remote}, C_{Local}) \cdot W_{Private} + C_{Local} \cdot W_{Private} \cdot (W_{Depth} - 1) \tag{4.12}$$

The time to write the private data is the same except that it is the last private volume, not the first, that is written to the home storage server. In actuality, this last volume is actually written first to the local network and then later extracted to the home storage server; however the model abstracts this and ignores the initial write to the local network. Formally, we define this as

$$T_{PrivateWrite} = C_{Local} \cdot W_{Private} \cdot (W_{Depth} - 1) + BW(C_{Remote}, C_{Local}) \cdot W_{Private} \tag{4.13}$$

Note that there are actually $W_{Depth} + 1$ private volumes but of these only the first W_{Depth} are read and only the last W_{Depth} are written. The first private volume is read and not written and the last is written but not read; only private volumes between a parent and child job are both read and then later written during the execution of the workload. Notice also that the equations for the estimated time for both $T_{PrivateRead}$ and $T_{PrivateWrite}$ are identical except for the ordering of the addends.

To predict the total compute time, $T_{Compute}$, for the All allocation we multiply the average compute time W_{Run} by the number of jobs in each pipeline, W_{Depth}, or mathematically

$$T_{Compute} = W_{Run} \cdot W_{Depth} \tag{4.14}$$

Because the All allocation does not require any barriers, all jobs can run as soon as their job dependencies are satisfied. Because no jobs must wait to execute, even though there may exist a large degree of runtime variability, W_{Var}, this runtime variability is *not* considered within the predictive model.

Warm phase

The number of warm phases, V_{Warm}, is relative to the total number of expected concurrently executing pipelines (V_{Exec}), the total number of compute nodes (C_{CPU}), and the total number of pipelines (W_{Width}). Mathematically, we define the number of warm phases to be

$$V_{Warm} = \left\lceil \frac{W_{Width}}{min(V_{Exec}, C_{CPU})} \right\rceil - 1 \tag{4.15}$$

which subtracts the initial warm phase from the total number of phases that are required to run a total of W_{Width} pipelines at a concurrency of either V_{Exec} when sufficient compute nodes are available or at C_{CPU} when they are

not.

Remember that for the All allocation the expected number of executing pipelines in the steady state is always the full width of the workload because there exist no data constraints, therefore, for the All allocation:

$$V_{Exec} = W_{Width} \tag{4.16}$$

Each warm phase consists of the time to fetch the batch data locally ($T_{WarmBatch}$) plus the same times we previously defined to read the private input ($T_{PrivateRead}$), to write the private output ($T_{PrivateWrite}$), and to compute ($T_{Compute}$):

$$T_{WarmPhase} = T_{WarmBatch} + T_{PrivateRead} + T_{PrivateWrite} + T_{Compute} \tag{4.17}$$

Therefore the last estimate needed to predict the runtime for an All allocation is $T_{WarmBatch}$, the time to fetch the batch data in the warm phase after it has already been cached in the local cluster. This is computed as the product of the local bandwidth (C_{Local}) and the total amount of batch data ($D_{TotBatch}$):

$$T_{WarmBatch} = C_{Local} \cdot D_{TotBatch} \tag{4.18}$$

This reveals another simplification within the model in that it does not consider that the batch data is actually striped across the local cluster such that for job j computing on node m reading the batch data, one stripe will not actually be fetched from the local network because that stripe is already hosted on the compute machine m. This same simplification is true for the reading and writing of private data.

4.4.2 Predicting Runtime for AllPrivate

The algorithm for predicting the runtime for workloads scheduled using an AllPrivate allocation is shown in Figure 4.11. There is only *one* difference between the predictive model for the AllPrivate allocation and the All allocation.

This sole difference results from the use of barriers in an AllPrivate allocation which changes the predicted value we set for $T_{Compute}$. Because each job may need to wait for its siblings jobs to complete, the mean job time, W_{Run}, is not of interest here; rather we use an incremental value between the mean time and a crude estimate of the longested predicted job time ($W_{Run} + W_{Run} \cdot W_{Var}$).

We weight this by approximating the likelihood of barriers through determining the maximum number of batches volumes which can be concurrently allocated, V_{Batch}. In other words, the $T_{Compute}$ values increases for decreasing values of V_{Batch}. Mathematically, we set $T_{Compute}$ as:

$$T_{Compute} = (W_{Run} + \frac{W_{Run} \cdot W_{Var}}{V_{Batch}}) \cdot W_{Depth} \tag{4.19}$$

and V_{Batch} is determined mathematically as:

$$V_{Batch} = \left\lfloor \frac{C_{Storage} - D_{TotPrivate}}{W_{Batch}} \right\rfloor \tag{4.20}$$

Note that the model does not assume anything about the actual runtime distribution of the workload. Even

$$
\begin{aligned}
T_{Total} &= T_{ColdPhase} + V_{Warm} \cdot T_{WarmPhase} \\
T_{ColdPhase} &= T_{ColdBatch} + T_{PrivateRead} + T_{PrivateWrite} + T_{Compute} \\
T_{ColdBatch} &= BW(C_{Remote}, C_{Local}) \cdot D_{TotBatch} \\
BW(C_{Remote}, C_{Local}) &= \frac{1}{\frac{1}{C_{Remote}} + \frac{1}{C_{Local}}} \\
D_{TotBatch} &= W_{Depth} \cdot W_{Batch} \\
T_{PrivateRead} &= BW(C_{Remote}, C_{Local}) \cdot W_{Private} + C_{Local} \cdot W_{Private} \cdot (W_{Depth} - 1) \\
T_{PrivateWrite} &= C_{Local} \cdot W_{Private} \cdot (W_{Depth} - 1) + BW(C_{Remote}, C_{Local}) \cdot W_{Private} \\
T_{Compute} &= (W_{Run} + \frac{W_{Run} \cdot W_{Var}}{V_{Batch}}) \cdot W_{Depth} \\
V_{Batch} &= \left\lfloor \frac{C_{Storage} - D_{TotPrivate}}{W_{Batch}} \right\rfloor \\
D_{TotPrivate} &= (W_{Depth}+1) \cdot W_{Private} \cdot W_{Width} \\
V_{Warm} &= \left\lceil \frac{W_{Width}}{min(V_{Exec}, C_{CPU})} \right\rceil - 1 \\
V_{Exec} &= W_{Width} \\
T_{WarmPhase} &= T_{WarmBatch} + T_{PrivateRead} + T_{PrivateWrite} + T_{Compute} \\
T_{WarmBatch} &= C_{Local} \cdot D_{TotBatch}
\end{aligned}
$$

Figure 4.11: **The Predictive Model for the AllPrivate Allocation.** *These equations form the predictive model which the scheduler uses to predict the total time to completion for workloads scheduled using the AllPrivate allocation. Only the darkened equations differ from those already defined in Figure 4.10 for the All allocation. Here the only difference is in the prediction for the compute time which considers runtime variation which is ignored in the All allocation.*

$$
\begin{aligned}
T_{Total} &= T_{ColdPhase} + V_{Warm} \cdot T_{WarmPhase} \\
T_{ColdPhase} &= T_{ColdBatch} + T_{PrivateRead} + T_{PrivateWrite} + T_{Compute} \\
T_{ColdBatch} &= BW(C_{Remote}, C_{Local}) \cdot D_{TotBatch} \\
BW(C_{Remote}, C_{Local}) &= \frac{1}{\frac{1}{C_{Remote}} + \frac{1}{C_{Local}}} \\
D_{TotBatch} &= W_{Depth} \cdot W_{Batch} \\
T_{PrivateRead} &= BW(C_{Remote}, C_{Local}) \cdot W_{Private} + C_{Local} \cdot W_{Private} \cdot (W_{Depth} - 1) \\
T_{PrivateWrite} &= C_{Local} \cdot W_{Private} \cdot (W_{Depth} - 1) + BW(C_{Remote}, C_{Local}) \cdot W_{Private} \\
T_{Compute} &= (W_{Run} + \frac{W_{Run} \cdot W_{Var}}{V_{Batch}}) \cdot W_{Depth} \\
V_{Warm} &= \left\lceil \frac{W_{Width}}{min(V_{Exec}, C_{CPU})} \right\rceil - 1 \\
V_{Exec} &= \left\lfloor \frac{C_{Storage} - D_{TotBatch}}{2W_{Private}} \right\rfloor \\
T_{WarmPhase} &= T_{WarmBatch} + T_{PrivateRead} + T_{PrivateWrite} + T_{Compute} \\
T_{WarmBatch} &= C_{Local} \cdot D_{TotBatch}
\end{aligned}
$$

Figure 4.12: **The Predictive Model for the AllBatch Allocation.** *These equations form the predictive model which the scheduler uses to predict the total time to completion for workloads scheduled using the AllBatch allocation. Only the darkened equations differ from those already defined in Figure 4.10 for the All allocation. The sole difference is in the prediction for the expected degree of concurrency,* V_{Exec}.

though the runtimes within our synthetic workloads fit a normal distribution, the model is concerned only with estimating the cost of barriers and does so with this crude estimate of the runtime of the longest running job in the set. Again, our models are concerned only with relative, and not absolute accuracy, and so we feel this simplification is reasonable in that it achieves the desired behavior by penalizing allocations that allow barriers relative to their likelihood of encountering them.

4.4.3 Predicting Runtime for AllBatch

The algorithm for predicting the total runtime for workloads scheduled using an AllBatch allocation is similar to that of All, and is shown in Figure 4.12 with the sole difference found between the formulae for computing the estimated number of concurrently executabling pipelines (V_{Exec}).

Although the predictive algorithms are mostly identical, this does *not* mean that there is no difference between the allocations. Remember, as was shown in Figure 4.9, that each of these allocations is possible for different sizes of batch and private volumes relative to the total available storage. As was seen previously, the area in which the All allocation is possible is a strict subset of the area in which the AllBatch allocation is possible.

When both All and AllBatch allocations are possible, the number of executable pipelines will be the full width of the workload (*i.e.* $V_{Exec} = W_{Width}$) and the predicted runtimes will be identical. The difference however is that the AllBatch allocation is possible for larger values of batch and private volume sizes when the All allocation is not possible. At this point, the maximum utilization for the AllBatch allocation will drop to the lower value which is equal to the number of jobs which can have access to both their input and output private volumes after allocating

all batch data:

$$V_{Exec} = \left\lfloor \frac{C_{Storage} - D_{TotBatch}}{2W_{Private}} \right\rfloor \tag{4.21}$$

One interesting thing to note here is that although the predictive algorithms are very similar, the anticipated workload traversal for an AllBatch allocation as opposed to either an All or AllPrivate allocation may be almost completely opposite. Remember when we looked at their respective workflow traversals in Figures 4.6, 4.4, and 4.5, that the AllBatch allocation follows a depth-first traversal, the AllPrivate allocation proceeds in a breadth-first manner, and being unconstrained, the All traversal is free to do either. How then with such different traversals can these disparate scheduling allocations have such similar predictive algorithms?

The answer is that, although each allocation may result in a differently ordered traversal, the total set of the components of each of these traversals is the same: they are merely ordered differently. For example, the AllPrivate traversal first does the cold phase for the jobs at $W_{Depth} = 1$, then does the warm phases, if necessary, for the remainder of the jobs at that depth. Only then does it repeat this for each subsequent level of depth. Conversely, the AllBatch allocation does *all* of the cold phases first for each depth, and only then does all of the warm phases.

To risk belaboring the point, imagine a workload of depth two which requires one warm phase for both an AllPrivate and an AllBatch allocation. The total predicted runtime for the AllPrivate allocation is

$$T_{ColdPhase}(W_{Depth} = 1) + T_{WarmPhase}(W_{Depth} = 1)$$

$$+T_{ColdPhase}(W_{Depth} = 2) + T_{WarmPhase}(W_{Depth} = 2)$$

whereas for the AllBatch allocation it is

$$T_{ColdPhase}(W_{Depth} = 1) + T_{ColdPhase}(W_{Depth} = 2)$$

$$+T_{WarmPhase}(W_{Depth} = 1) + T_{WarmPhase}(W_{Depth} = 2)$$

Notice that the AllPrivate allocation alternates between cold and warm phases whereas the AllBatch allocation does all cold phases and then does all the warm phases. We see here therefore that the individual components are the same and only the ordering is different. For an All allocation the components are also the same but the ordering of the actual traversal is arbitrary as it can execute without any data constraint.

4.4.4 Predicting Runtime for Slice

The predictive algorithm to determine the anticipated runtime of a workload using a Slice allocation is very similar to that used for the AllPrivate allocation and is shown in Figure 4.13. Although their predictive models are similar, the AllPrivate and Slice allocations are certainly not identical. There are two main differences.

First, Slice does also use runtime variability to compute its expected compute time, $T_{Compute}$. However, this value is weighted by the maximum number of batch volumes which can be concurrently allocated, V_{Batch}, and this value is derived differently for Slice than it is for AllPrivate. For the Slice allocation, V_{Batch} is computed as the number of batches that fit in remaining storage after allocating a single volume for each pipeline and a second

$$
\begin{aligned}
T_{Total} &= T_{ColdPhase} + V_{Warm} \cdot T_{WarmPhase} \\
T_{ColdPhase} &= T_{ColdBatch} + T_{PrivateRead} + T_{PrivateWrite} + T_{Compute} \\
T_{ColdBatch} &= BW(C_{Remote}, C_{Local}) \cdot D_{TotBatch} \\
BW(C_{Remote}, C_{Local}) &= \frac{1}{\frac{1}{C_{Remote}} + \frac{1}{C_{Local}}} \\
D_{TotBatch} &= W_{Depth} \cdot W_{Batch} \\
T_{PrivateRead} &= BW(C_{Remote}, C_{Local}) \cdot W_{Private} + C_{Local} \cdot W_{Private} \cdot (W_{Depth} - 1) \\
T_{PrivateWrite} &= C_{Local} \cdot W_{Private} \cdot (W_{Depth} - 1) + BW(C_{Remote}, C_{Local}) \cdot W_{Private} \\
T_{Compute} &= (W_{Run} + \frac{W_{Run} \cdot W_{Var}}{V_{Batch}}) \cdot W_{Depth} \\
V_{Batch} &= \left\lfloor \frac{C_{Storage} - W_{Private} \cdot W_{Width} - V_{Exec} W_{Private}}{W_{Batch}} \right\rfloor \\
V_{Exec} &= \left\lfloor \frac{C_{Storage} - W_{Batch} - W_{Private} \cdot W_{Width}}{W_{Private}} \right\rfloor \\
V_{Warm} &= \left\lceil \frac{W_{Width}}{min(V_{ExecAve}, C_{CPU})} \right\rceil - 1 \\
V_{ExecAve} &= \frac{n \cdot n + (D_{NumJobs} - 2n) \cdot V_{Exec} + n \cdot n}{D_{NumJobs}} \\
n &= \left\lfloor \frac{C_{Storage} - W_{Batch}}{2 W_{Private}} \right\rfloor \\
D_{NumJobs} &= W_{Depth} \cdot W_{Width} \\
T_{WarmPhase} &= T_{WarmBatch} + T_{PrivateRead} + T_{PrivateWrite} + T_{Compute} \\
T_{WarmBatch} &= C_{Local} \cdot D_{TotBatch}
\end{aligned}
$$

Figure 4.13: **The Predictive Model for the Slice Allocation.** *These equations form the predictive model which the scheduler uses to predict the total time to completion for workloads scheduled using the Slice allocation. Only the darkened equations differ from those already defined in Figure 4.11 for the AllPrivate allocation. The difference between the two predictive algorithms is due to the phase behavior in the Slice allocation in which a greater number of jobs can execute concurrently at the beginning and the end of the workflow than during the middle.*

volume for as many additional pipelines as can execute. Mathematically, we determine this as so:

$$V_{Batch} = \left\lfloor \frac{C_{Storage} - W_{Private} \cdot W_{Width} - V_{Exec} W_{Private}}{W_{Batch}} \right\rfloor \tag{4.22}$$

Note that because the Slice allocation aggressively executes as many pipelines as possible after allocating a single batch, V_{Batch} in a Slice allocation will usually be only 1 except for very small batch volume sizes or when storage is not scarce.

Second, Slice uses a more complex set of equations for estimating the average concurrency of the workflow than does AllPrivate. As was shown in Figure 4.9, the total area in which the AllPrivate allocation is possible is a subset of the possible area for the Slice allocation. Because it only allocates storage for a single horizontal slice of the workload, the Slice allocation is possible for workloads of greater depths than is the AllPrivate allocation. When both are possible however the predicted runtimes and workflow traversals are indeed identical.

However, at the point at which the AllPrivate allocation is no longer possible due to the size of the volumes relative to the total amount of available storage, the runtime prediction for the Slice allocation becomes more complex. While the AllPrivate allocation is possible, all private volumes can be concurrently allocated and there are no concurrency constraints.

As the total amount of private volume increases and the AllPrivate allocation becomes no longer possible, the Slice allocation remains possible but full concurrency will no longer be possible and concurrency limits may be imposed depending on the number of available compute nodes, C_{CPU}.

The expected number of executable pipelines, V_{Exec} for the Slice allocation is as the number of pipelines that can execute (*i.e.* have access to a second volume) after allocating all the data necessary to hold a single horizontal slice of the workload or mathematically as

$$V_{Exec} = \left\lfloor \frac{C_{Storage} - W_{Batch} - W_{Private} \cdot W_{Width}}{W_{Private}} \right\rfloor \tag{4.23}$$

This equation is correct but in order to predict the runtime for the Slice allocation, we need to take into consideration that there are periods during the execution of the workflow for a Slice allocation in which the number of executing pipelines can exceed the expected steady state value.

This effect was seen during our earlier examination in of the workflow traversal for Slice as shown in Figure 4.7. As was observed then, the maximum number of concurrently executing pipelines may be greater at both the very beginning of the workflow as well as the end. During the middle portion of the workflow, while every pipeline has at least one private volume allocated, then the workflow executes at the steady state concurrency. This difference is due to the fact that the storage needed to allocate a horizontal slice of the workload is smaller at the very beginning and end of the workflow than it is in the middle.

Therefore, to estimate the runtime, we then need to compute the *average* concurrency between these maximum and steady state values. This complexity was ignored for the other allocations because for them the number of executing pipelines is more consistent throughout the traversal of the workload.

Because of this effect, and as will be seen more clearly in our evaluation, the width and the depth of the workload influence the average concurrency of the workload. For small widths and depths, this additional concurrency at the beginning and end of the workflow is a larger influence, as width or depth increase, this influence is amortized.

$$
\begin{aligned}
T_{Total} &= \sum_{i=1}^{V_{Cycles}} T_{Total}(Slice, SubWorkload_i) \\
V_{Cycles} &= \left\lceil \frac{W_{Width}}{V_{Exec}} \right\rceil \\
V_{Exec} &= \left\lfloor \frac{C_{Storage} - W_{Batch}}{2W_{Private}} \right\rfloor \\
W_{Width}(SubWorkload_i) &= \begin{cases} V_{Exec}, & i < V_{Cycles} \\ W_{Width} \bmod V_{Exec}, & i = V_{Cycles} \end{cases}
\end{aligned}
$$

Figure 4.14: **The Predictive Model for the Minimal Allocation.** *These equations form the predictive model which the scheduler uses to predict the total time to completion for workloads scheduled using the Minimal allocation. This predictive algorithm splits the workload into multiple sub-workloads and predicts the runtime for each sub-workload using the predictive model for the Slice allocation as shown in Figure 4.13.*

The exact effect seen is that the first and last n jobs can all execute concurrently where n is strictly greater than the number of jobs expected to concurrently execute in the steady state, V_{Exec}. This value of n is found by determining how many pipelines can have both their input and output private volumes allocated after allocating a single batch volume (as we will see, this is the same formula used to determine the expected concurrency for the Minimal allocation):

$$n = \left\lfloor \frac{C_{Storage} - W_{Batch}}{2W_{Private}} \right\rfloor \tag{4.24}$$

The reason for this, and we will see this again in Section 4.4.5 when we discuss the predictive runtime algorithm for the Minimal allocation, is that the Minimal allocation acts similarly to the Slice allocation with the difference being that it is willing to refetch batch data in order to maximize concurrency as opposed to the Slice allocation which accepts reduced concurrency in order to avoid refetching batch data. Indeed at the very beginning and end of its workflow traversal, when not all of the private volumes are allocated, the Slice allocation acts exactly as the Minimal allocation does during its entire traversal.

Therefore, we estimate the average expected concurrency for the Slice allocation to be the average of n and V_{Exec} weighted by the number of jobs to be executed at each of these two concurrencies. The number of jobs to be executed at a concurrency of n is n at the beginning plus another n at the end. The remainder of the jobs ($W_{Width}W_{Depth} - 2n$) will be executed at the expected steady state for Slice, V_{Exec}. Actually, some number of jobs will execute at a concurrency *between* these two values as the workload transitions between these phases but this effect is ignored in the model.

Mathematically, we reduce all of the above into the following equation to compute the expected average concurrency, $V_{ExecAve}$, for the Slice allocation:

$$V_{ExecAve} = \frac{n \cdot n + (D_{NumJobs} - 2n) \cdot V_{Exec} + n \cdot n}{D_{NumJobs}} \tag{4.25}$$

4.4.5 Predicting Runtime for Minimal

Predicting the runtime for a workload scheduled using a Minimal allocation is almost identical to the prediction using a Slice allocation with one major difference. Effectively, the Minimal allocation does a series of Slice allocations for vertical subsets of the workload (*i.e.* some number of pipelines, p, less than the width of the workload, W_{Width}); each of these vertical subsets can then be executed using the Slice allocation at full CPU utilization (*i.e.* when $V_{Exec} >= C_{CPU}$). Notice that in the base case in which the Slice allocation can itself execute at full CPU utilization, the number of vertical subsets for the Minimal allocation will be one, and the behavior (and runtimes) of the two allocations will be the same.

Because the Minimal allocation aggressively frees batch data allocations to maximize CPU utilization, it effectively maximizes CPU utilization at the possible expense of refetching batch data over the WAN. Conversely, the Slice allocation minimizes WAN traffic by executing all jobs that read from a batch volume before removing that volume and in so doing may reduce CPU utilization.

Notice that our model assumes that batch volumes (and in fact all volumes) cannot be partially removed. Although our synthetic workloads do read batch data only once, many real applications reread (and rewrite) data at various phases of their execution as discussed in Chapter 2. For this reason, it is difficult for a batch scheduler to anticipate when a particular volume can be removed during the execution of the job. Only when a job completes, can a volume be safely removed.

Each vertical subset of the workload executed using a Minimal allocation will therefore refetch the batch data in its entirety and as such the predicted runtime for each vertical subset will be the predicted runtime for a slice allocation for that subset. We then multiply by the predicted number of subsets to find the total estimated runtime for the Minimal allocation.

The number of subsets and therefore the number of cycles of the Slice allocation, V_{Cycles}, is found by dividing the total number of pipelines, W_{Width}, by the expected concurrency, V_{Exec}, or mathematically as:

$$V_{Cycles} = \left\lceil \frac{W_{Width}}{V_{Exec}} \right\rceil \tag{4.26}$$

The expected concurrency for the Minimal allocation is found by determining how many pipelines can have concurrent access to both their input and output private volumes after allocating a single batch volume, which is expressed mathematically as:

$$V_{Exec} = \left\lfloor \frac{C_{Storage} - W_{Batch}}{2W_{Private}} \right\rfloor \tag{4.27}$$

We then create V_{Cycles} smaller sub-workloads, each identical to the complete workload but with the smaller width V_{Exec}. Notice that the width of the final sub-workload may be smaller as it is the remainder of the pipelines after executing the first $V_{Cycles} - 1$ sub-workloads,

$$W_{Width}(Subworkload_{V_{Cycles}}) = W_{Width} \bmod V_{Exec} \tag{4.28}$$

The predicted runtime for the Minimal allocation is then the sum of the predicted runtime for each of these n

```
total_runtime = 0
num_pipes     = workload_width
warm          = 0
WHILE ( num_pipes )
    runtime       = predictRuntime( num_pipes, warm )
    total_runtime += runtime
    num_pipes     = runtime * failure_rate
    IF ( allocation EQ All OR allocation EQ AllBatch )
        warm = 1
    FI
END
```

Figure 4.15: **Modelling the Effect of Failure.** *This figure contains the pseudo-code for our failure model. Notice that failure is modelled almost identically across the different scheduling allocations. The difference is that for those allocations which continue to cache batch data after it is used (All and AllBatch) the failure model assumes that the batch data needed for rescheduled failed pipelines is already cached.*

smaller sub-workloads using a Slice allocation, or mathmetically:

$$T_{Total} = \sum_{i=1}^{V_{Cycles}} T_{Total}(Slice, SubWorkload_i) \tag{4.29}$$

The complete predictive algorithm for the Minimal allocation is summarized in Figure 4.14.

4.4.6 Predicting the Effect of Failure

The observant reader may have noticed that our predictive algorithms use only ten of the eleven workload and environment characteristics. The algorithms account for the number of compute nodes (C_{CPU}), for the amount of storage ($C_{Storage}$), for the remote and local bandwidths (C_{Remote} and C_{Local}), for the width and depth of the workload (W_{Width} and W_{Depth}), for the size of the batch and private volumes (W_{Batch} and $W_{Private}$), and finally for the compute time and its variability (W_{Run} and W_{Var}).

The failure rate, $C_{Failure}$, of the compute environment is *not* considered within any of the individual predictive algorithms. Rather, it is considered as an external effect and is modelled *almost* identically across each of the algorithms.

Specifically, to estimate the effect of failure, we take the predicted runtime of the workload and multiple it by the failure rate to determine the number of failed pipelines. We then estimate the runtime of a new workload whose width is the number of failed pipelines from the previous. We repeat until no pipelines are expected to fail. The estimated runtime is therefore the sum of the runtimes of each workload run with a diminishing number of pipelines.

The way in which the failure model is applied differently across the different allocations is in regards to batch data. When we reschedule the failed pipelines for those allocations which do not proactively remove batch volumes (*i.e.* All and AllBatch), we adjust the model such that it does not run any "cold" cycles but rather assumes that all batch data is already cached. Conversely, for the other allocations, AllPrivate, Slice, and Minimal, which remove the batch data as they descend through the workflow, the failure model correctly accounts that all batch data must be refetched from the home node for the rescheduled failed pipelines. The pseudo-code for our failure model is

	W_{Width}	W_{Depth}	Total Batch	$\frac{W_{Batch}}{C_{Storage}}$	Total Private	$\frac{W_{Private}}{C_{Storage}}$	W_{Run}	W_{Var}
batch	350	5	883 GB	70.6%	420 GB	0.1%	5000s	500s
private	350	5	150 GB	12.0%	5250 GB	1.0%	5000s	500s
mixed	350	5	225 GB	18.0%	1050 GB	0.2%	5000s	500s

Table 4.4: **Baseline Characteristics of the Synthetic Workloads.** *This table provides the baseline values describing each of our three canonical batch-pipeline workloads.*

shown in Figure 4.15.

4.4.7 Winnowing the Allocations

To evaluate the accuracy of our predictive model we examine the performance of each allocation across the range of workload and environmental characteristics which we have identified as having an affect on performance (the top two groups of rows in Table 4.1). To simplify this evaluation we remove the All and AllPrivate allocations from consideration.

The All allocation is not interesting here because of the complete lack of constraint. Because the entire set of volumes is allocable whenever All is possible, the scheduler in such a case need make no allocation decisions. This problem then reverts to the already addressed problem in batch computing of making job placement decisions in regards to available computational resources [69, 119, 109]. As the problem we are examining is the more difficult challenge of coordinating both the CPU and the data allocations, we are not interested in cases in which the data allocation is trivially solvable. Further, All is a subset of AllBatch; indeed in every instance in which All is possible, the behavior of AllBatch and All are the same.

The AllPrivate allocation is similarly uninteresting. Intuitively, and also as evidenced in Figure 4.9, AllPrivate is a strict subset of Slice. There exists no situation is which AllPrivate is possible but Slice is not; further in every situation in which AllPrivate is possible, Slice is a preferable allocation strategy because AllPrivate wastes storage space holding already consumed private volumes whereas Slice releases these volumes thereby allowing other volumes to be allocated. In many cases, there is no difference between these allocations but in some, this extra space allows Slice to hold more data and may allow it to descend more quickly through the workload and avoid barriers imposed in AllPrivate.

In regards to failure, AllPrivate might seem preferable because it could allow pipelines which experience failed jobs to rollback partially instead of entirely. However, for a canonical workload in which all private volumes are the same size, any storage space being used for a shallower private volume would be strictly better utilized to hold a deeper volume instead. For example, imagine a pipeline executing in a high-failure environment. Storing a backup copy of a private volume from some depth d for this pipeline allows it to resume from that point should it fail. When the pipeline progresses deeper to a depth of $d + 1$, any storage holding the backup volume from d would be strictly better utilized to hold a copy of the volume from $d + 1$; in other words, resuming a pipeline at depth $d + 1$ is strictly better than resuming at depth d.

4.4.8 Synthetic Workloads

Having eliminated All and AllPrivate as uninteresting allocation strategies, our selection decision now must consider only the three remaining, AllBatch, Slice and Minimal. To examine the differing performance profiles of

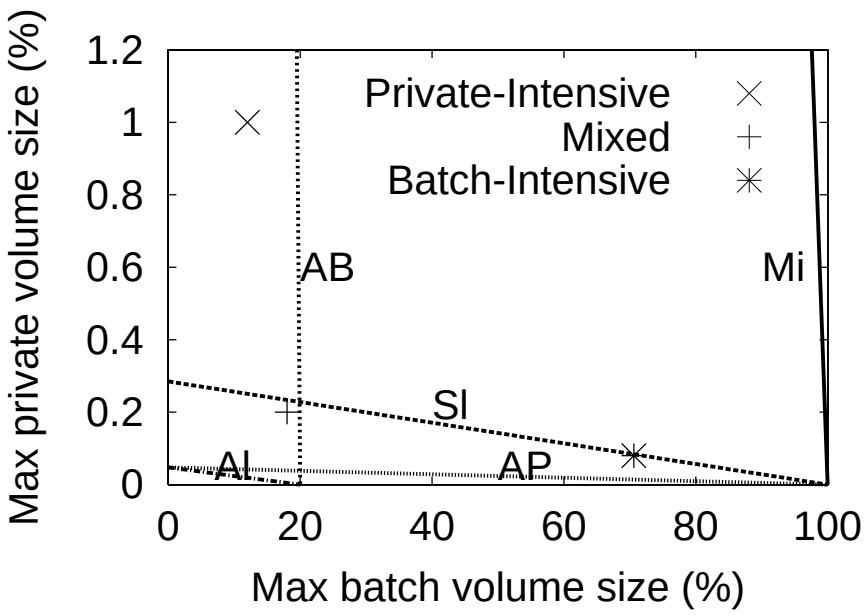


Figure 4.16: **Possible Allocations for the Synthetic Workloads.** *This graph displays the position of each of our three synthetic workloads within the possible areas for our different scheduling allocations. Notice that all three allocations are possible for only the mixed workload, for the batch-intensive workload, only the Slice and Minimal allocations are possible and for the pipe-intensive workload, only the AllBatch and Minimal allocations are possible.*

these allocation strategies and to evaluate the accuracy of our predictive model, we exercise these allocations for each of three representative workloads across the range of workload and environmental characteristics as defined in the top two groups of Table 4.1.

The three synthetic workloads that we use are based on our profiling observations from Chapter 2 and are constructed such that one is *batch-intensive*, one is *private-intensive* and the third is *mixed*, being neither batch- nor private-dominant but having middling values for each. The precise values used in each of these workloads are shown in Table 4.4. For these workloads, we normalize the amount of compute time across each of them and select a value such that when running a single instance of each pipeline locally at the home storage server, the pipeline would be performing I/O for approximately 50% of its total execution time.

As shown in Figure 4.16, we choose these workloads such that for the batch-intensive workload, only the Slice and Minimal allocations are possible, for the private-intensive workload only the AllBatch and Minimal allocations are possible and finally for the mixed workload, the AllBatch, Slice, and Minimal allocations are possible but an All allocation and an AllPrivate allocation are not.

Finally shown in Table 4.5 are the values used to describe the computational environment. Remember as stated earlier in this chapter that these six constants describing the workload and these five constants describing the environment comprise the entirety of the information needed by the scheduler to make predictive estimates so long as the workload is canonical and the compute infrastructure is homogeneous.

4.4.9 Allocation Behavior

To illustrate the different behaviors exhibited by the three allocation strategies of AllBatch, Slice, and Minimal, we show in Figures 4.17, 4.18 and 4.19 the storage and CPU allocation profiles for each of the scheduling

Number of compute nodes	C_{CPU}	50
Total available storage	$C_{Storage}$	250 GB
Failure rate	$C_{Failure}$	0.0
Remote bandwidth	C_{Remote}	12 MB/s
Local bandwidth	C_{Local}	1 MB/s

Table 4.5: **Baseline Characteristics of the Compute Environment.** *This table lists the five constants used as the baseline characteristics of our compute environment. Our evaluation section includes experiments in which we vary different combinations of these values.*

allocations for the three synthetic workloads. We focus our discussion here on the mixed workload in Figure 4.19 because it is the only one of the three workloads for which all of the three allocation strategies are possible. For these graphs, the x-axis shows the elapsed execution time for the workload. For the graphs on the left, the y-axis is the percent of the cluster CPUs allocated and for the graphs on the right it is the percent of allocated cluster storage.

Notice initially that none of the three allocation strategies is able to completely utilize the available computation. There are two reason why this is so. For the AllBatch allocation, the complete allocation of the batch data leaves only enough available storage to execute on approximately half of the compute nodes (specifically for this case there are fifty compute nodes and the maximum number of concurrent pipelines for the AllBatch allocation is twenty-five).

For the other two allocations, Slice and Minimal the maximum number of concurrent pipelines reaches 100% but as seen here, this full utilization cannot be maintained throughout the entire execution of the workload. The vertical white stripes which indicate this lack of complete utilization are due to barriers imposed within the workload. These barriers are necessary because not all of the batch volumes can be simultaneously allocated. When some pipelines progress to a depth at which the necessary batch volume is not allocated and there is not available storage to do so, those pipelines must wait for other jobs to complete. The width of these barriers is directly related to the variability between compute times (W_{Var}).

Comparing vertically across the different allocation strategies, we see that the Slice allocation as shown in the middle realizes the highest throughput for the mixed workload finishing approximately 15% more quickly than the AllBatch allocation on top and 25% more quickly than Minimal on the bottom. Note also the different phase behaviors exhibited in the Slice and Minimal allocations but absent in AllBatch. Due to its simultaneous allocation of all batch volumes, the AllBatch allocation need impose no barriers during the execution of the workload. This results in a smoother allocation of both storage and CPU for the AllBatch allocation. Notice also that the maximum value for the CPU allocation in AllBatch is only half that of the other two but that it remains steady throughout its execution.

Notice also the difference in the phase behavior between the Slice and the Minimal allocations. Remember that the Minimal allocation effectively splits the workload into sub-workloads and schedules each using a Slice allocation. In this case, with $W_{Width} = 350$, the Minimal allocation will result in two separate phases (to execute across the entire width of the workload). Each of these two phases will then itself consist of five internal phases, one for each depth in the workload.

This also explains why the two phases for Minimal are skewed. The first phase executes 205 pipelines while the second executes only the remaining 145. The scheduler could choose to balance these phases but since there is no compelling reason to do so, it does not. In fact, as we will note later when we examine the effect of failure,

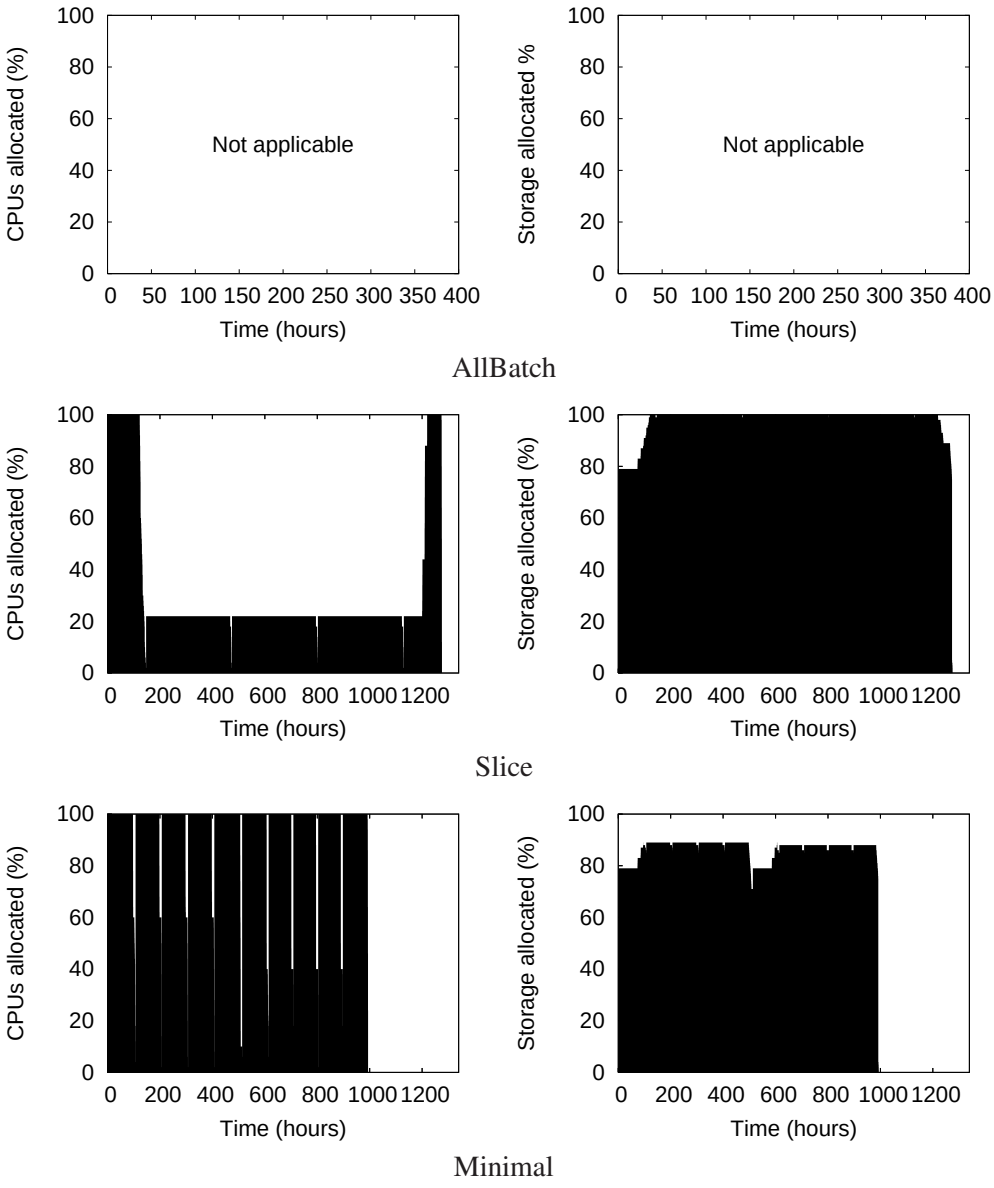


Figure 4.17: **The CPU and Storage Allocations for the batch-intensive Workload.** *These figures show the percent of CPUs allocated on the left and the percent of storage allocated on the right for each of our three different scheduling allocations when executing the synthetic batch-intensive workload. The top most graphs show the allocations for the AllBatch allocation, the middle for Slice, and the bottom for the Minimal allocation. In all graphs, the y-axis shows the percent allocated as a function of the elasped time. The graphs for the AllBatch allocation are blank and labeled as "Not applicable" here because the AllBatch allocation is not possible for the batch-intensive workload as the total amount of batch data in this workload exceeds the total amount of available storage.*

it proves advantageous to *not* balance such that failures within the initial "large" phases can be absorbed into the subsequent "smaller" ones without requiring entire additional phases.

One question that can arise while looking at the graphs for the mixed workload is why consider a Minimal allocation at all? For this experiment, it is clear that the Minimal allocation offers no advantages over Slice; it finishes more slowly, and it transfers twice as much batch data over the WAN (*i.e.* once for each phase). However, there are several advantages to Minimal. First, Minimal is possible in some situations as we will see below where Slice is not. Second, even in some situations where both are possible, as in the batch-intensive workload shown in Figure 4.17, Minimal can outperform Slice.

When Minimal is able to outperform Slice it is because it is able to achieve a higher degree of concurrency, V_{Exec}. The Slice allocation allocates an entire horizontal slice such that all jobs within that slice can execute before descending within the workload. It does this to avoid needing to refetch any batch data; all access to a batch volume is completed through the breadth-first traversal in a Slice allocation. Conversely, the Minimal allocation does not allocate the entire horizontal stripe but only allocates a subset such that each pipeline in the subset can immediately execute (*i.e.* it has both its input and output private volumes allocated).

For both the batch-intensive and the mixed workloads, the Minimal allocation has a higher expected concurrency. Specifically, for the mixed workload, the expected concurrency for the Slice allocation is only 60, whereas it is 205 for the Minimal. However, since the number of compute nodes (C_{CPU}) is set at 50, the benefit that Minimal receives from its ability to simultaneously execute a greater number of pipelines is not realized. However, in the batch-intensive workload, the expected concurrency for the Slice allocation is only 10 whereas it is over 100 for the Minimal allocation. As the number of available compute nodes is sufficient to allow each allocation to run at its fully expected concurrency, the increased utilization of the Minimal allocation allows it to complete approximately 20% more quickly than the Slice allocation as is seen in Figure 4.17.

Notice another interesting effect in the Slice allocation for the batch workload is that the concurrency is greater at the ends than in the middle. This is because the steady state utilization in Slice is constrained because storage is allocated for every executing pipeline. However, at the beginning of a workload as pipelines begin executing and again at the end of a workload as pipelines begin to finish, the total amount of storage required to hold all executing pipelines is smaller thereby allowing a larger number of pipelines concurrent access to the second private volume that they need in order to execute. Notice this is the visualization of the effect that was described earlier in our discussion of the predictive model for the Slice allocation.

Finally, we see for the private workload in Figure 4.18 that the AllBatch allocation executes the workload most quickly. Although the Minimal allocation is also possible here, it underperforms relative to AllBatch due to its barriers and to its refetch penalty.

4.5 Evaluation

Having crafted a set of representative workloads and a compute environment in which their execution is interesting due to having data constraints, we now examine the relative performance of the different allocation strategies as we vary each of the eleven variables defining the workloads and the environment.

We do not however individually examine each of these variables; some are evaluated as their ratio to another, such that we now examine eight experiments derived from our eleven variables. The width of the workload, W_{Width}, is evaluated as a ratio to the number of compute nodes, C_{CPU}. The depth of the workload, W_{Depth},

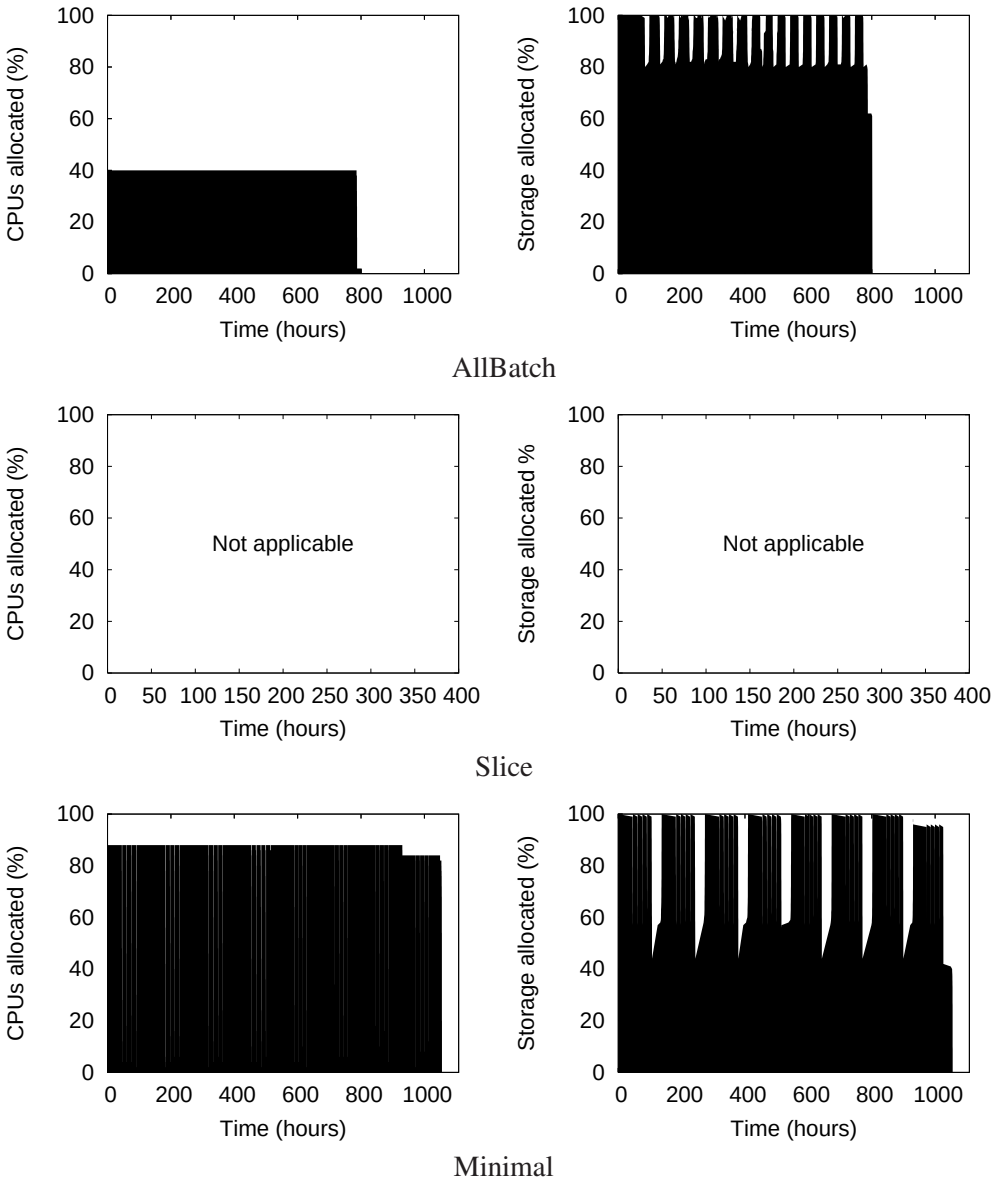


Figure 4.18: **The CPU and Storage Allocations for the private-intensive Workload.** *These figures show the percent of CPUs allocated on the left and the percent of storage allocated on the right for each of our three different scheduling allocations when executing the synthetic private-intensive workload. The top most graphs show the allocations for the AllBatch allocation, the middle for Slice, and the bottom for the Minimal allocation. In all graphs, the y-axis shows the percent allocated as a function of the elasped time. The graphs for the Slice allocation are blank and labeled as "Not applicable" here because the Slice allocation is not possible for the private-intensive workload as the total amount of data needed to allocate a horizontal slice of this workload exceeds the total amount of available storage.*

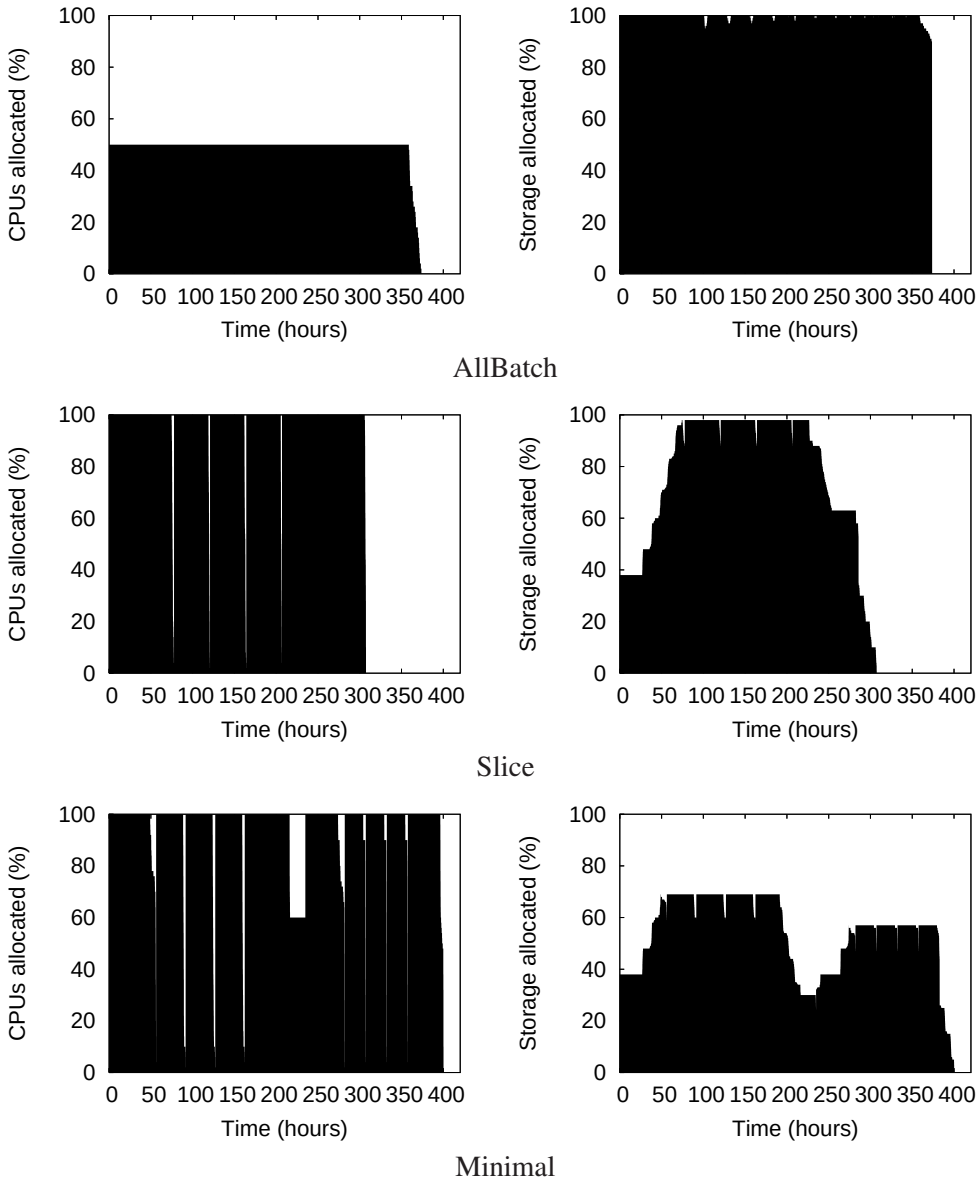


Figure 4.19: **The CPU and Storage Allocations for the mixed Workload.** *These figures show the percent of CPUs allocated on the left and the percent of storage allocated on the right for each of our three different scheduling allocations when executing the synthetic mixed workload. The top most graphs show the allocations for the AllBatch allocation, the middle for Slice, and the bottom for the Minimal allocation. In all graphs, the y-axis shows the percent allocated as a function of the elasped time.*

is evaluated in isolation. The size of batch volumes, W_{Batch}, is evaluated in relation to the amount of available storage, $C_{Storage}$, as is also the case for the size of private volumes, $W_{Private}$. Compute times, W_{Run}, are evaluated in isolation and compute time variability, W_{Var}, is evaluated in relation to compute time, W_{Run}. The network bandwidths, C_{Remote} and C_{Local}, are evaluated in relation to each other, and finally the failure rate, $C_{Failure}$, is evaluated in isolation.

4.5.1 Sensitivity to Workload Width

Shown in Figure 4.20 is the effect that increasing the width of the workload has on the performance of the three studied allocation strategies. The left most graphs are the effect on the batch workload, the middle on the private and the right most on the mixed workload. Within each set of graphs, the top row shows the total CPU utilization during the execution of the workload. This number is computed by taking the total amount of compute time consumed within the workload and dividing it by the product of the total number of CPUs and the total runtime of the workload. For example, for a workload that uses only a single CPU at a time and runs on a cluster of ten compute nodes, the total CPU utilization achieved would be 10%. The next row shows the total amount of data transferred between the home storage server and the compute cluster. The third row shows the achieved throughput for each allocation. These first three rows for the CPU utilization, the wide-area network traffic, and the throughput are all measurements of the simulated workloads.

Conversely, in the fourth row, we show the throughput numbers as estimated by our predictive model. Finally, in the fifth and bottom row, we quantify the predictive accuracy of our model. This bottom row shows the runtime of the predicted high throughput allocation normalized against the runtime of the actual observed high throughput allocation drawn with a thick dashed line. In cases in which the model correctly identifies the high throughput (*i.e.* low runtime) allocation, this value is zero. Also shown for comparison is the normalized value of the *worst* possible allocation to the "best."

Note that we are *not* comparing the predicted throughput of the model to the actual throughput of the simulator; rather we use the relative values from the model to select which allocation is expected to achieve the highest throughput. We then compare that expected allocation's actual simulated runtime to the runtime of the simulated allocation empirically observed to achieve the highest throughput.

For all these graphs, the x-axis is not the absolute value of W_{Width} but is expressed rather as the ratio to the number of cluster compute nodes (*i.e.* $W_{Width} : C_{CPU}$). There are several things to notice in these graphs. First, we'll discuss why each of the allocations behaves as it does for each of these workloads and then we'll discuss the ability of our model to correctly predict these behaviors.

Looking at Figure 4.20 we see throughput crossover points for both the batch and the mixed workloads. These crossovers occur because for "thin" workloads, the Slice allocation is able to concurrently execute on all of the compute nodes (*i.e.* $V_{Exec} \geq C_{CPU}$). For the batch-intensive workload, the AllBatch allocation is not possible and the Minimal allocation underperforms Slice due to the redundant use of the WAN as shown in the right most graph. However, as the width of the workload increases, there is a time in which the Slice allocation remains possible but only at a drastically reduced value of V_{Exec}. At this point, the throughput of the Minimal allocation surpasses that of Slice. A similar effect is shown for the mixed workload yet in this case the AllBatch allocation is possible albeit at a lower value of V_{Exec}. Notice further that as expected from Table 4.3 both AllBatch and Minimal are relatively robust to increasing workload widths while Slice is significantly more sensitive. For the private-intensive workload, no crossover effect is revealed because the Slice allocation is not possible for this workload and as noted

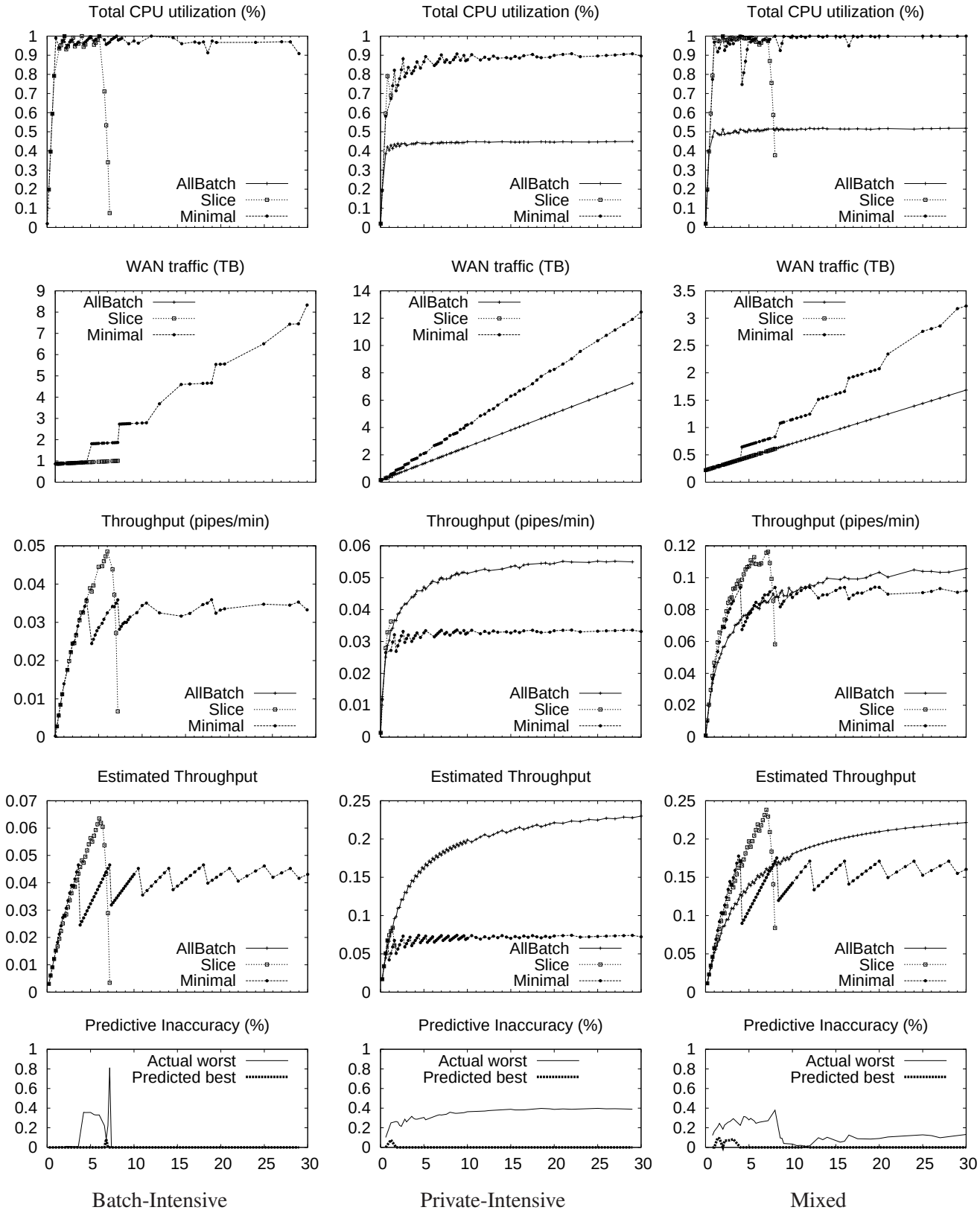


Figure 4.20: **Sensitivity to Workload Width (x-axis is $W_{Width} : C_{CPU}$).**

previously, the other allocations are relatively constant across increasing workload widths.

Notice also in these graphs that the measurements sometimes abruptly end; this can be see clearly for the throughput of the Slice allocation for the mixed workload (the middle graph on the right). Notice that the throughput here drops between values five and ten on the x-axis and then entirely disappears. The reason for this is that at a certain width, the Slice allocation is entirely no longer possible.

The sawtooth pattern exhibited by the Minimal allocation is due to the "tail" effect. For workloads in which the expected number of concurrent pipelines (V_{Exec}) is not an even factor of the total workload width, the last group of jobs to execute will be fewer than the previous groups. Looking at the batch workload for example, V_{Exec} is approximately three times that of C_{CPU} such that it achieves peaks in throughput and CPU utilization at multiples of three along the x-axis.

As regards to the predictive accuracy, note that the model is not absolutely accurate especially in regards to its predicted throughput values. Although close for the batch workload, it is off by a factor of five for the private and a factor of two for the mixed. However, it is not our intention that the model be perfectly accurate. Indeed in the interests of simplification, we have ignored many system aspects in the model such as network contention, the memory subsystems of the compute nodes and the home storage server as well as the latencies for the disks and networks. These simplifications cause the model to overestimate the throughput of the workload.

However, our focus is not on making absolute predictions but is rather on selecting from the possible allocation strategies. Therefore, our interest is in the relative throughputs between the different allocations. In this regard, our model mirrors closely the simulated results. In particular, to evaluate the predictive ability of the model to identify the "best" allocation, we examine the crossover points in these graphs. Notice that there is a perfect one-to-one correlation in the crossover points between the simulated and the modeled results. Further that each pair of correlated crossover points occur at relatively the same position on the x-axis; in other words, our model correctly predicts both that a crossover will occur as well as where it will occur.

This predictive accuracy is seen qualitatively in the graphs along the bottom row of Figure 4.20. Points at which the model mispredicts the highest throughput allocation appear as a positive value in these graphs. The height of these points is the percent of additional (unnecessary) runtime that would be incurred by a scheduler using this misprediction as compared to a "magic" scheduler which always correctly identifies the highest throughput allocation.

In other words, the model does not *always* select the "best" allocation. However, as long as it only makes mistakes when the relative throughput of its expected best allocation is close to the actual best allocation, then the effect of this inaccuracy is minimized. Notice further that although the model does not make perfect estimates as to the predicted throughput of the allocations, it does provide perfect information as to which allocations are and are not possible. Thus, for every case in which only one allocation is possible, the model will correctly identify it. Only when multiple allocations are possible does there exist any possibility of misprediction. Notice further that the model consistently avoids making "bad" mispredictions. Although for this experiment the worst possible prediction sometimes is eighty percent worse than the best, the model never makes mispredictions greater than ten percent.

For a dynamically changing environment as is typical in batch computing, these predictive models do not need to be optimal. In fact searching for optimal information in a batch system is often quixotic as the information can become stale and therefore sub-optimal very quickly. For this reason, it becomes much more important not necessarily to select the "best" strategy but to avoid very bad ones. *In all cases, our model does so.*

4.5.2 Sensitivity to Workload Depth

Similar to varying the W_{Width} is varying the W_{Depth} as is shown in Figure 4.21. As expected from Table 4.3, the performance of the AllBatch allocation is highly sensitive to the depth. As the depth increases, the total amount of batch data also increases thereby allowing less storage for fewer pipelines to execute (*i.e.* the increase in W_{Depth} results in a decrease in V_{Exec} for the AllBatch allocation).

Some less obvious results are also seen in these experiments. Notice in the batch workload that there exists a crossover point between the Slice and Minimal allocations and that these throughput results relate closely to the utilization. For small depths, the Slice allocation can achieve higher levels of concurrency than at greater depths. The reason for this discrepancy is that the V_{Exec} for Slice is different at the beginning and end of the workload's execution than it is in the middle. Remember that this same phenomenon was also seen earlier in Figure 4.17 and was discussed in our predictive model for the Slice allocation in Section 4.4.4.

In the middle of the execution, the Slice allocation must hold at least one private volume for every pipeline in the workload. However, at the beginning it need only hold one private volume for each pipeline that has already begun executing thereby allowing more pipelines to concurrently execute until they have all begun to execute. Similarly at the end of the execution as pipelines finish and are removed, more space becomes available and a greater number of pipelines can execute concurrently. Clearly then this effect is dependent on the depth and the relative benefit diminishes as the depth increases.

Notice again our predictive accuracy is very high; in this case never exceeding five percent of the normalized highest possible throughput. Although our absolute predictions are inaccurate, we correctly identify every crossover point and do so at almost their precise location on the x-axis.

There is one inaccuracy in the model that is worth mentioning here even though it does not result in a large amount of predictive inaccuracy. Notice that for both the batch-intensive and the private-intensive workloads, the model predicts increasing throughputs for the Minimal allocation as the depth of the workload increases. The reason for this prediction is that as the depth of a pipeline increases, the relative amount of private data that need traverse the wide-area network decreases as only the first and last private volumes need do so; the rest are localized and consumed in their entirety within the local network of the compute cluster. Therefore, as depth increases, we expect to achieve higher throughputs as a larger percentage of the private data is accessed at the higher local bandwidth, C_{Local}, than at the remote bandwidth to the home storage server, C_{Remote}.

For the private workload we get this correct. For the batch workload, however, this predicted effect is not seen; in fact, we predict a rising and then leveling throughput slope for the Minimal allocation when the actual slope as observed in our simulation drops and then levels. The reason for this is seen in the CPU utilization graph (the top left most graph). Notice that as the depth increases, there is a slight drop in the CPU utilization for the Minimal allocation. The model does not correctly anticipate this drop. The more radical drop in the Slice allocation, which is due to the effect that Slice achieves higher utilization at the very beginning and end of its scheduled, is correctly anticipated however.

This unpredicted drop in utilization for the Minimal allocation is due to compute time variability (W_{Var}) and its effect on the duration of barriers. For these experiments, the model assumes that the variability is constant and it is indeed set as a constant value (10%) in these experiments. However, as the number of jobs increases due to the increasing depth, the actual measured variability increases. With a normal distribution as are our compute times for our synthetic workloads, an increasing number of samples will continue to increase the measured minimum and maximum values. We see this effect here; with a small number of jobs, the actual variability is less and the

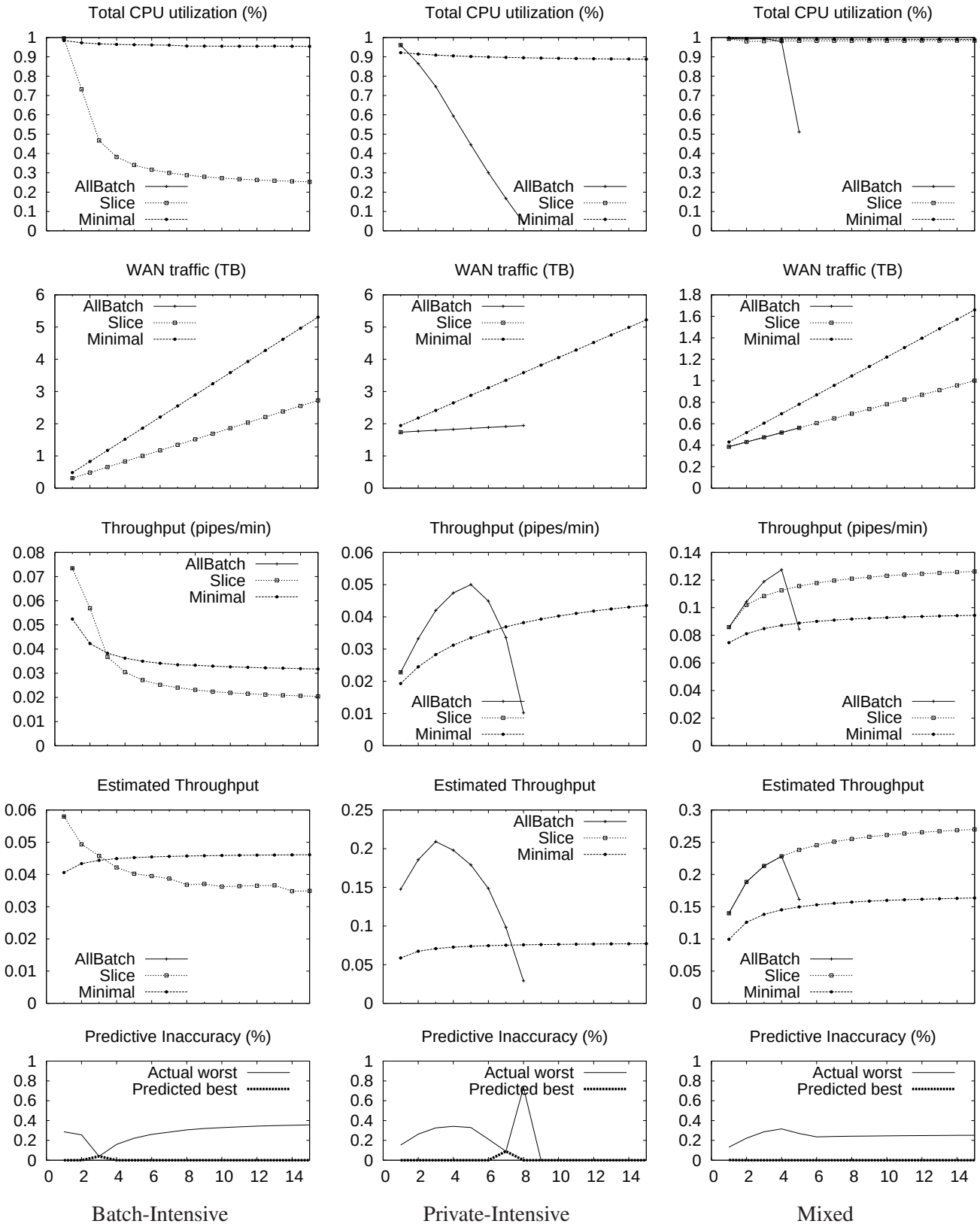


Figure 4.21: **Sensitivity to Workload Depth (x-axis is W_{Depth}).**

lost utilization to barriers is small; as jobs increase, so does variability leading to longer barriers, and we see the effect on throughput here. Notice however that although the model does not account for this effect, it still manages to never exceed five percent inaccuracy.

4.5.3 Sensitivity to Batch Volume Size

We also examine the effect of the batch volume size (W_{Batch}) on the achieved performance for the different allocation strategies. These results are shown in Figure 4.22; notice that the x-axis for these graphs is not the absolute value for W_{Batch} but rather the ratio of $W_{Batch} : C_{Storage}$. Because we vary here one of the aspects that distinguishes the three workloads, we effectively homogenize them such that the results as compared across workloads are more similar.

As expected, increasing the size of the batch volumes has the largest effect on the AllBatch allocation. The other allocations are effected but to a lesser degree as they need allocate only a single batch volume at a time. Notice however that after the AllBatch allocation is no longer possible for large values of W_{Batch}, there still exist interesting relative performances between the Slice and Minimal allocations. For the batch workload, the Slice allocation is possible long after the AllBatch allocation is not and while it remains possible it outperforms the Minimal allocation due to the heavy cost in the batch-intensive workload of transferring redundant data over the WAN. Yet ultimately, Minimal is able to continue after Slice is not. The effect is similar in the mixed workload but Slice becomes impossible more quickly here due to the larger sizes of private volumes in this workload as compared to the batch workload.

Here the behavioral predictions of the model are particularly striking in their accuracy. Notice the predicted slopes and crossover points are visually very similar. Qualitatively observing the measured values in the bottom graphs, we see again that the model never predicts worse than ten percent and successfully avoids making "bad" predictions.

4.5.4 Sensitivity to Private Volume Size

The experiment shown in Figure 4.23 is similar to those shown in Figure 4.22 because they also vary a parameter which tends to homogenize the three workloads although here it is the size of the private volumes ($W_{Private}$) rather than that of the batch (W_{Batch}). However, due to the distinction between batch and private volumes that a batch volume is shared across multiple pipelines while each pipelines accesses it own unique private volumes, this experiment is not able to extend as far on the x-axis relative to $C_{Storage}$ as does the experiment in which we vary W_{Batch}.

The observation to take away from these graphs is the overwhelming influence of the private volume size for all of the possible allocations studied (and for those not studied as well). Clearly and intuitively, the Slice allocation will be sensitive to increasing sizes of private volumes as it allocates each private volume accessed at a particular depth in the workload. Less obvious is the significant influence that private volume size has on the other allocations as well. For the AllBatch allocation, the increasing size of private volumes quickly reduces the number of pipelines that can concurrently execute (V_{Exec}). This effect is seen as well even in the Minimal allocation although it is slightly more robust.

Notice in this experiment, that not only does the model predict visually similarly sloping lines, but that it achieves here perfect accuracy, never making a single mispredictions.

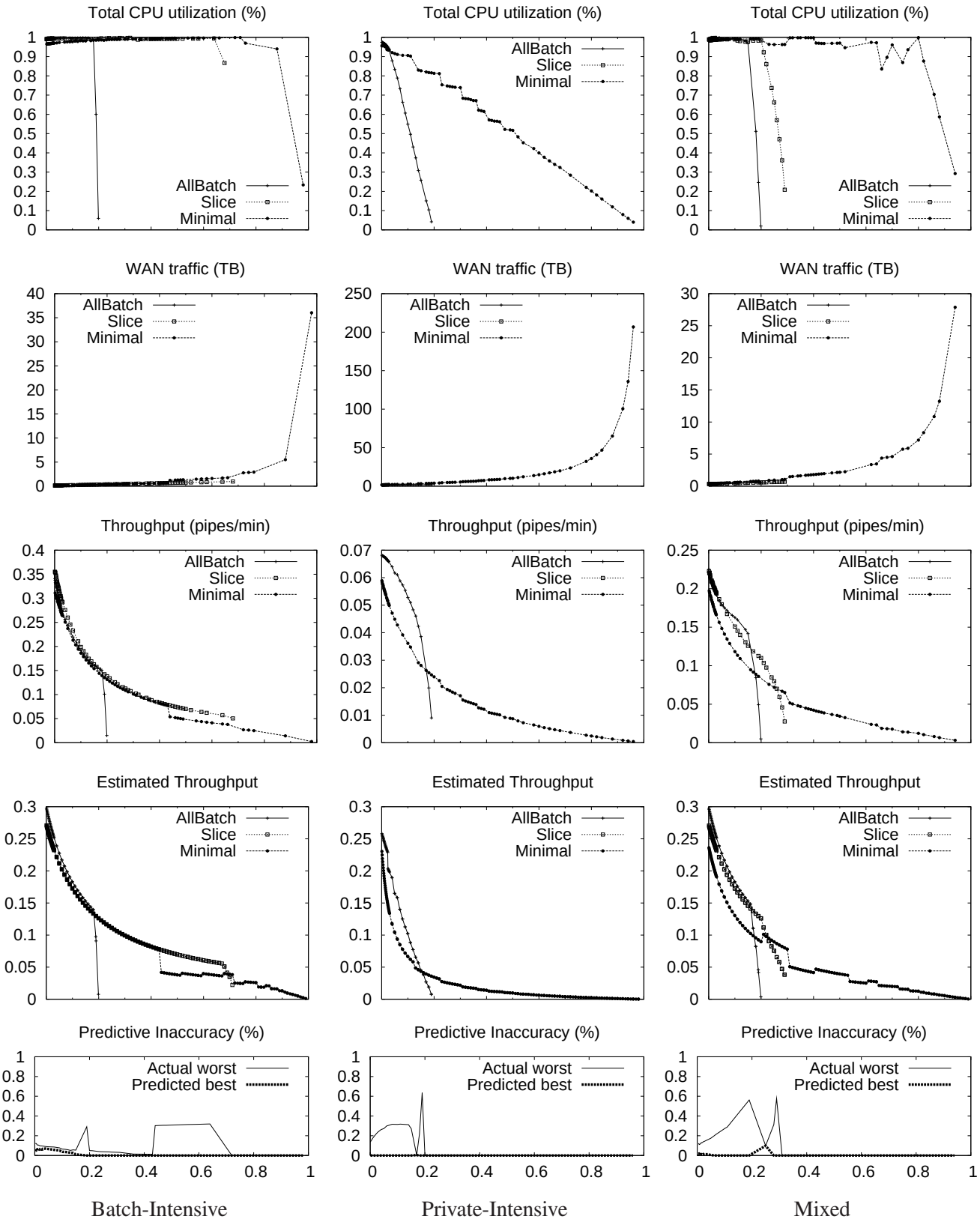


Figure 4.22: **Sensitivity to Batch Volume Size (x-axis is $W_{Batch} : C_{Storage}$).**

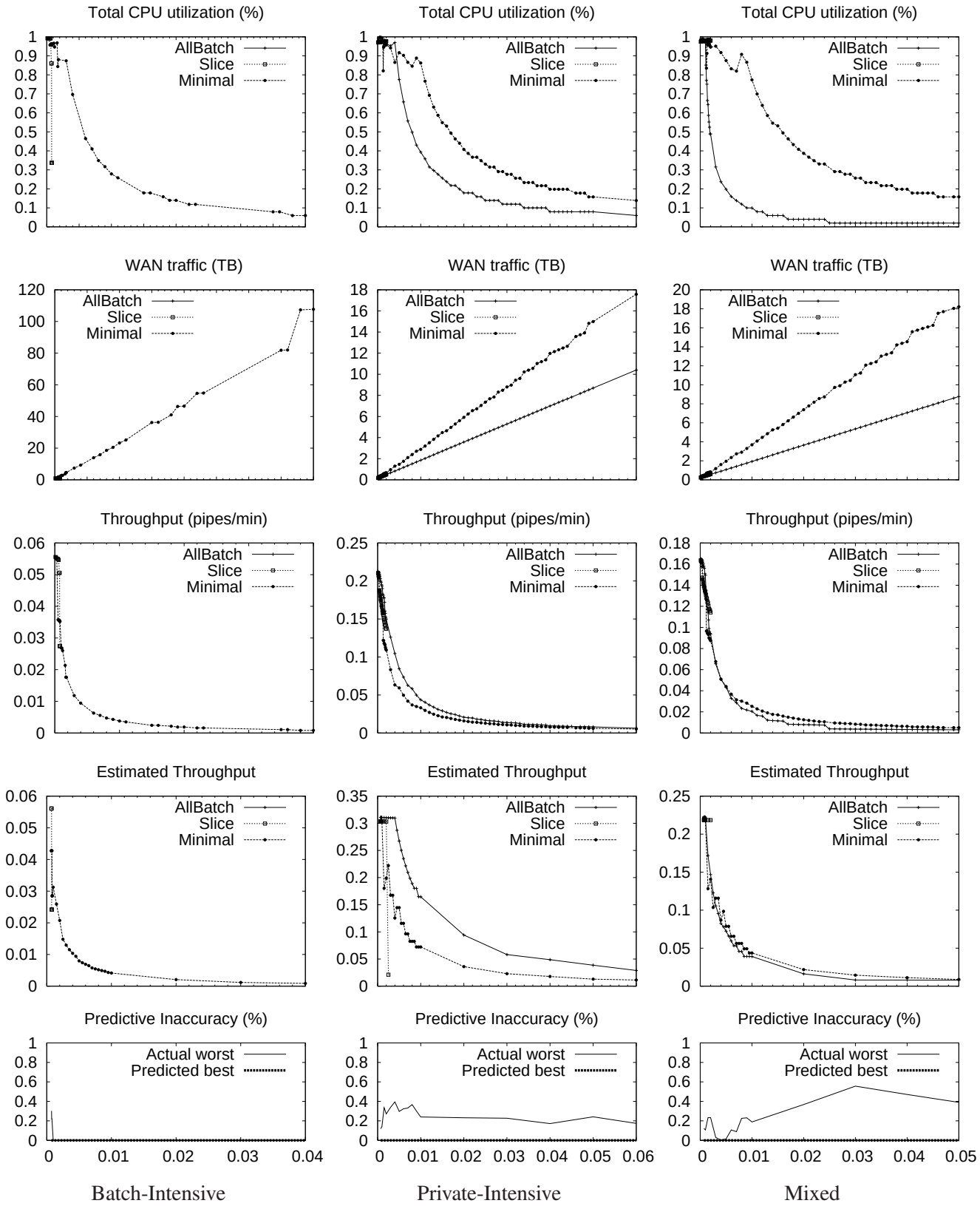


Figure 4.23: **Sensitivity to Private Volume Size (x-axis is $W_{Private} : C_{Storage}$).**

4.5.5 Sensitivity to Computation Time

The amount of time spent within each job doing computation and not I/O is another workload characteristic that affects the relative performance of the different allocation strategies as shown in Figure 4.24. Notice that as opposed to some of the other characteristics, the change in W_{Run} produces smoother lines. This is because the other characteristics vary integral values such as the size of volumes which are then divided into some integral number of data blocks within the system. The W_{Run} value however can more easily approximate non-integral numbers.

Note here that the relative performance of the Slice and Minimal allocations are mostly constant. However, there do appear crossover points between the AllBatch and Slice allocations. As we consider the constraints faced by each, the reason for these crossover points is understood. The AllBatch allocation is constrained in this experiment by having V_{Exec} less than C_{CPU} such that it makes progress but underutilizes the CPUs. Conversely, the Minimal allocation for these workloads has a higher V_{Exec} but must refetch the batch data (*i.e.* it has a higher $V_{Refetch}$). As the compute time increases relative to the time it takes to refetch the batch data, the relative refetch penalty incurred by the Minimal allocation is reduced while the underutilization penalty incurred by AllBatch remains constant. Thus, for increasing values of W_{Run}, we observe that the performance of the Minimal allocation improves relative to that of the AllBatch.

For this experiment, the predictive accuracy of our model is somewhat diminished, in one case, making a misprediction approaching twenty percent. Although the model correctly identifies both crossover points observed here (between AllBatch and Minimal for the private-intensive workload and between AllBatch and Minimal for the mixed), it does not as precisely identify their location on the x-axis.

4.5.6 Sensitivity to Runtime Variability

Similar to the effect of changing the job compute time is changing the variability across job compute times. The job compute times are taken from a normal distribution around a mean compute time with some variance that in the base case is set at ten percent. For the experiment shown in Figure 4.25, we increase this variance along the x-axis. One interesting effect seen (and not considered in the model) is that in some cases the increased variance can actually improve the throughput of the workload. This is because the increased variance creates "fast" jobs which descend quickly through the pipeline thereby effectively pre-fetching deeper batch volumes. Without variance, the initial placement of pipelines will execute in synchrony and all will effectively access their batch data at wide-area network bandwidths. However, if the batch data has been pre-fetched by a faster sibling, then the subsequent pipelines can then access that data at local area network bandwidths.

In regards to our predictive accuracy, notice that although the model does not account for this prefetch effect, it nevertheless makes entirely correct predictions in regards to the relative throughputs of the allocations.

4.5.7 Sensitivity to Runtime Variability for CPU-intensive Workloads

Beyond that however these results are somewhat uninspiring due to the small influence that W_{Run} has on the overall runtime of each pipeline. Therefore, we repeat this experiment with modified workloads such that W_{Run} is a much larger relative value. As mentioned earlier and as is shown in Table 4.4, W_{Run} is set such that it comprises approximately half of the total execution time for a pipeline run locally at the home storage server. For

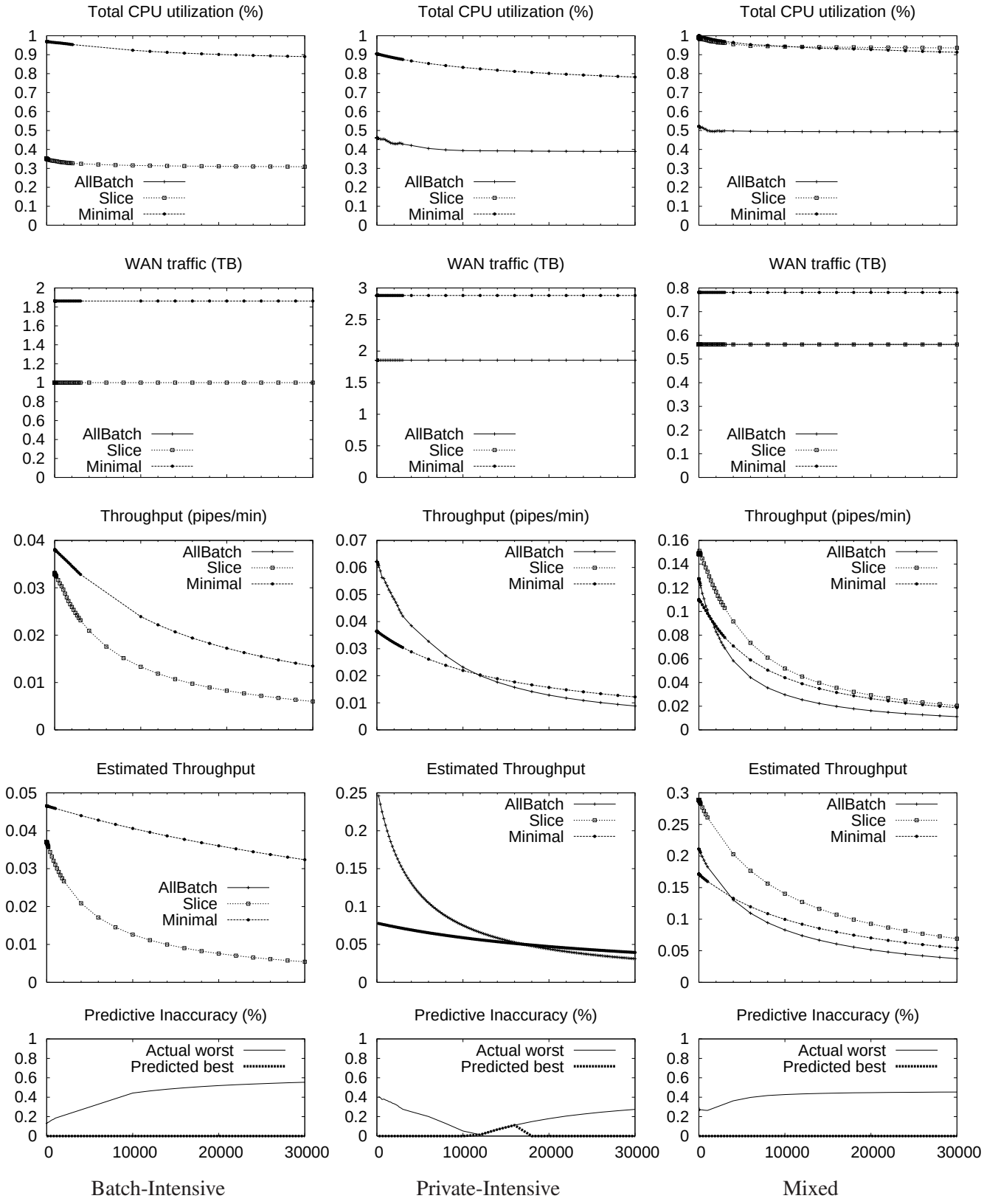


Figure 4.24: **Sensitivity to Compute Times (x-axis is W_{Run}).**

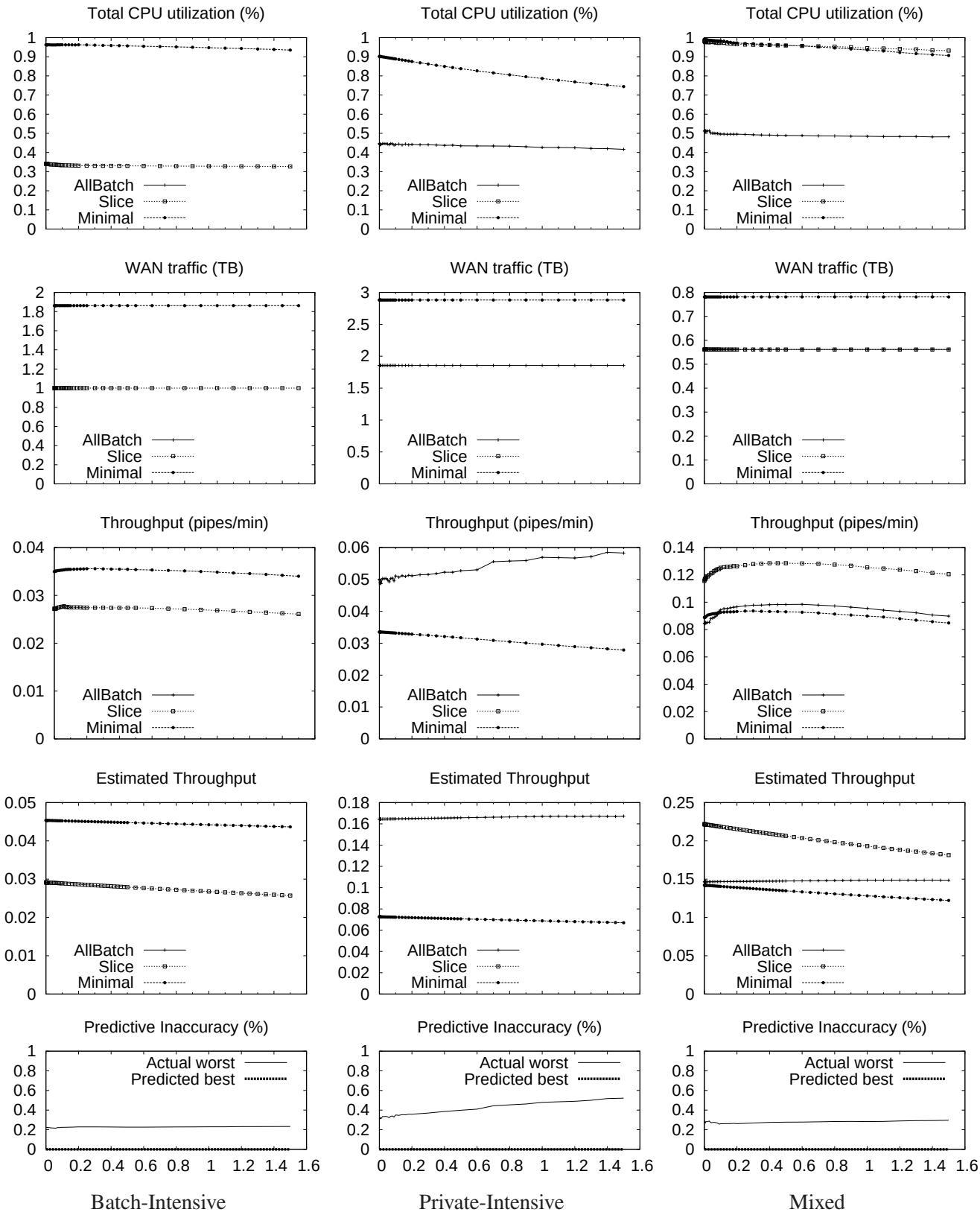


Figure 4.25: **Sensitivity to Runtime Variability (x-axis is $W_{Var} : W_{Run}$).**

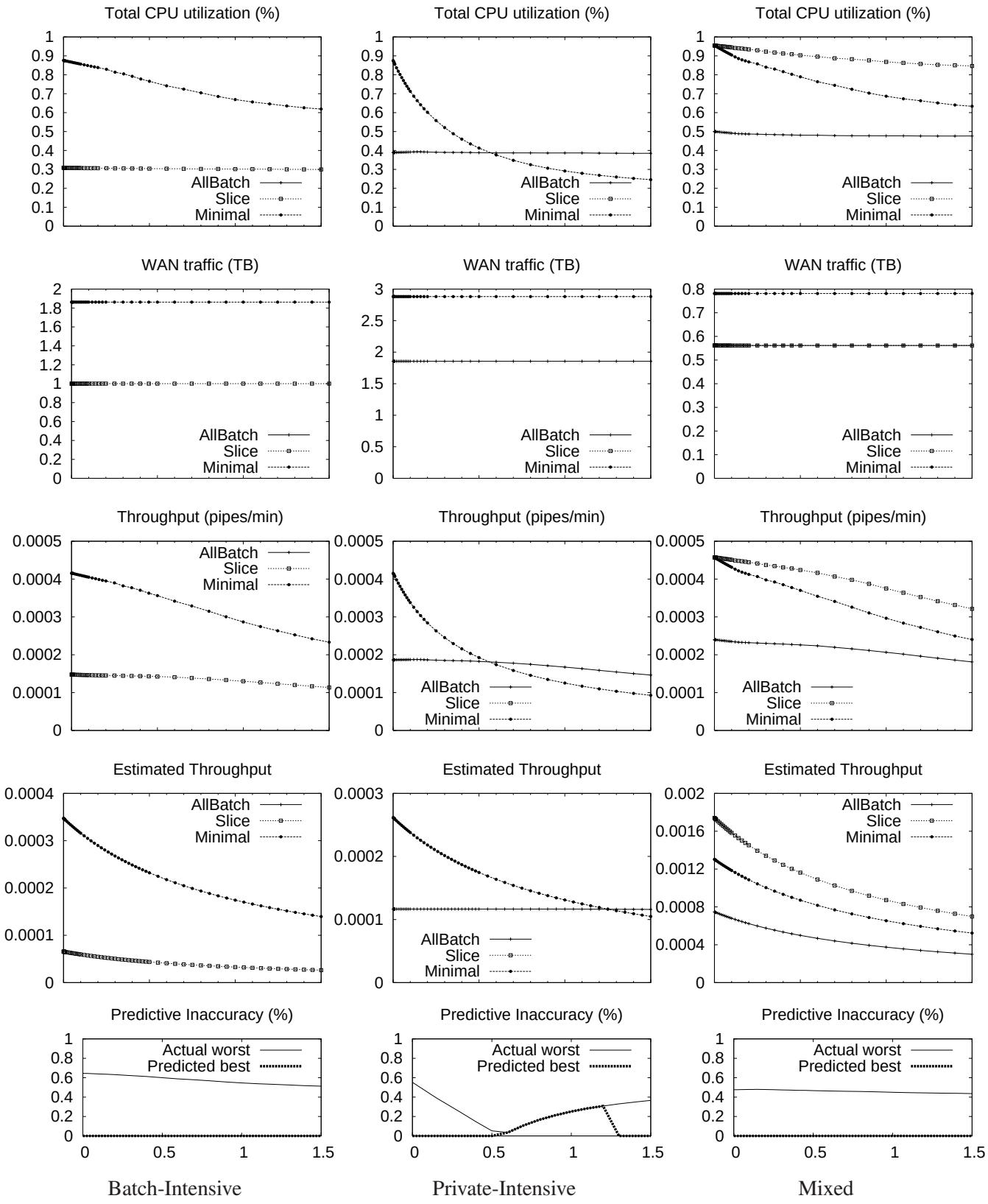


Figure 4.26: **Sensitivity to Runtime Variability for CPU Intensive Jobs (x-axis is $W_{Var} : (W_{Run} * 1000)$).**

these modified experiment, we increase C_{CPU} such that its relative proportion increases to approximately 99%. Specifically, in regards to Table 4.4, we set C_{CPU} to be five million seconds (a full 57 days).

With such a large value for C_{CPU} we can more readily see its influence on the different allocation strategies. As expected, a large variance should disproportionately penalize allocations which impose barriers. This is seen as the relative performance of the AllBatch allocation which imposes no barriers is mostly constant while the performance of the Minimal and Slice allocations which do impose barriers degrades much more rapidly. Notice further that the prefetch benefit as exhibited in the previous experiment is no longer seen here. With such large compute times relative to the time required to access the batch data, this prefetch benefit is negligible.

Here again, the accuracy of our model is somewhat diminished. Although it correctly identifies the crossover point, it misses the precise location on the x-axis. Although this miss results in approximately 30% difference in throughput, notice the model does successfully avoid making mispredictions with inaccuracy exceeding 50% in both the batch-intensive and private-intensive workloads.

4.5.8 Sensitivity to Network Bandwidths

The relative performance of the WAN and the LAN network also can influence the achievable throughputs of the various allocation strategies. Figure 4.27 shows this effect across the three different workloads. Intuitively as we consider each allocation's constraints, it is clear that the refetch penalty incurred by the Minimal allocation is most sensitive to this. This intuition is clearly substantiated in these results as the performance of Minimal relative to the other allocations improves more dramatically as the bandwidth to the remote storage server grows relative to the local bandwidth.

This intuition is modelled correctly as well. For this experiment, although possible mispredictions could result in throughput differences as high as 60%, our model consistently remains within three percent for the entirety of this experiment.

4.5.9 Sensitivity to Failure

Finally, the effect of failure ($C_{Failure}$) is examined in Figure 4.28 in which we increase the rate of failure along the x-axis. Notice that due to the randomness by which we induced simulated failure events, that these experiments are not deterministic as have been the previous. For that reason, we ran each point for thirty iterations and show here the means and the standard deviations. Another difference between these graphs and those shown previously is that here we include graphs for the numbers of recorded failures as well, and remove graphs for the wide-area network traffic; for the previous experiments, the baseline value for the failure rate was zero and thus there were no failures to be measured.

Each failure event "resets" a compute node in the system such that any running job on that compute node is evicted and all data contained on that node's disk and memory is similarly lost. Due to the implementation of the batch-aware distributed file system, these failure events wipe out a striped portion of the batch data as well as the entirety of the pipeline data held for the job running on that node. Therefore, we expect the failure penalty to disproportionately affect the Slice and the Minimal allocations because in addition to the progress lost in the executing pipeline, failures in these allocations require another refetch of the entire set of batch volumes. Conversely in the AllBatch allocation, failure results in a loss of progress in the executing pipeline but only a *stripe* of batch data and not the entire set. This behavior is seen most clearly in the mixed workload in which the

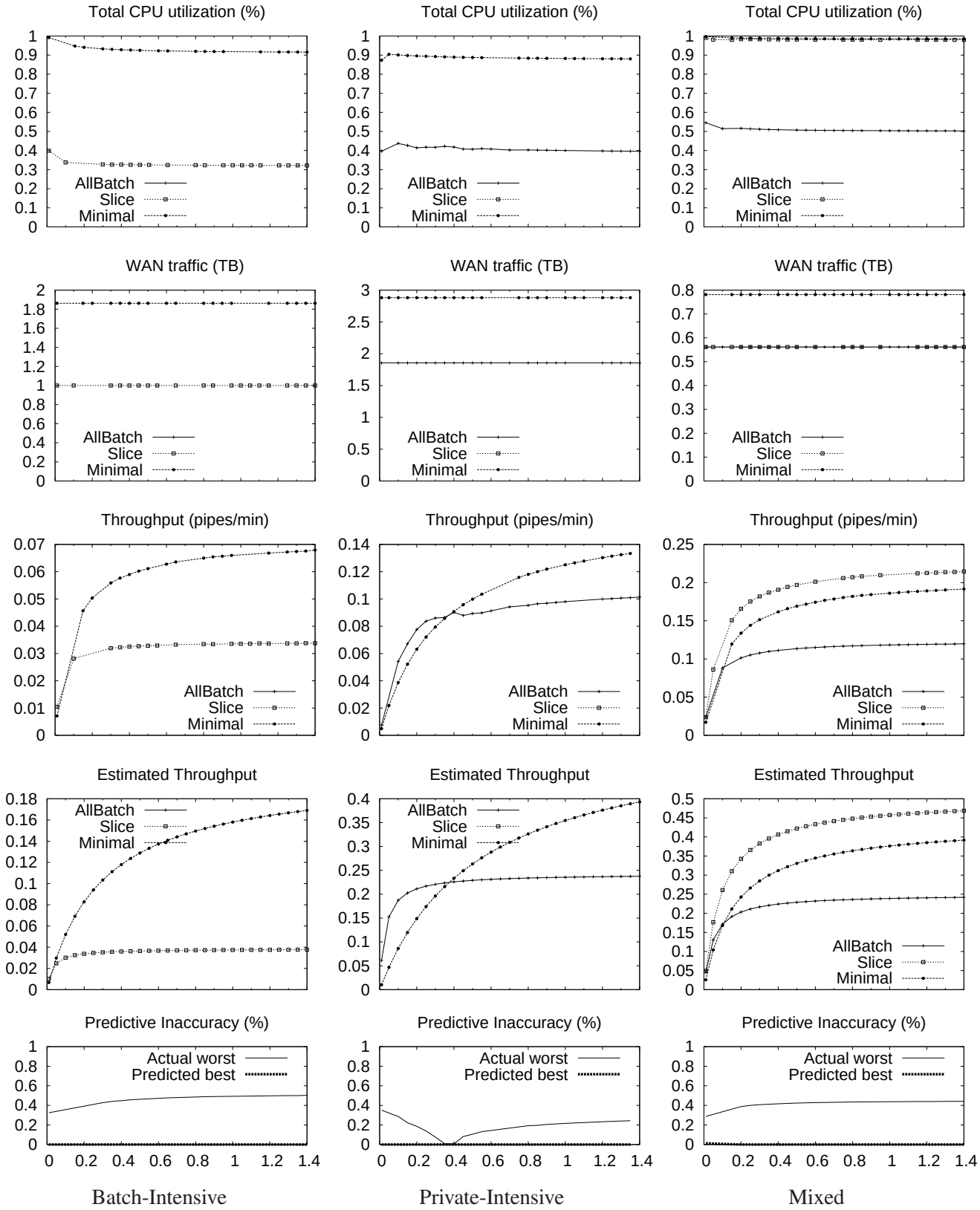


Figure 4.27: **Sensitivity to Remote Bandwidth (x-axis is $C_{Remote} : C_{Local}$).**

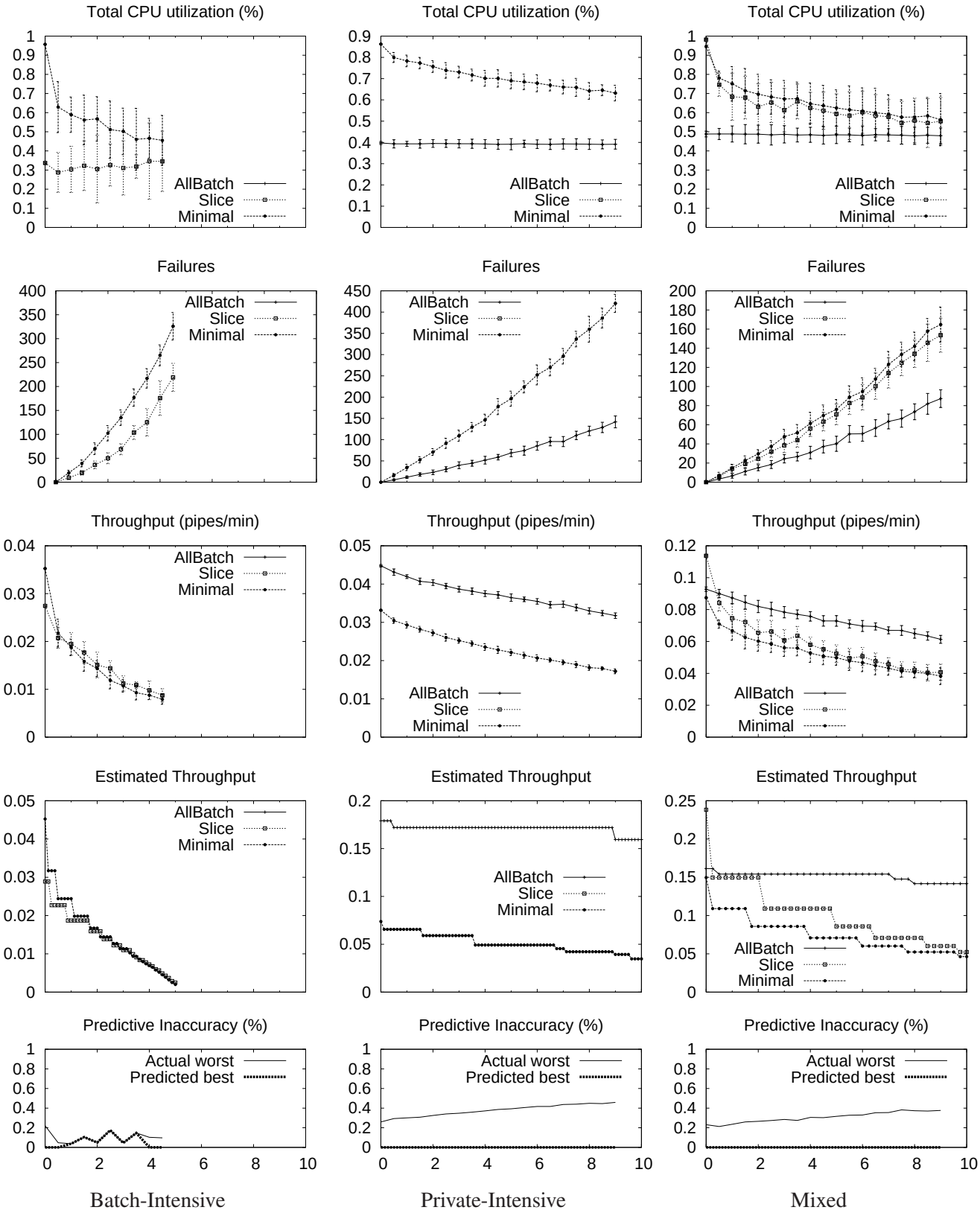


Figure 4.28: **Sensitivity to Failure Rate (x-axis is $C_{Failure}$).**

Slice allocation achieves the highest throughput for low rates of failure but then quickly drops below the AllBatch allocation as the rate of failure increases.

Although the predictive model does suffer some small inaccuracy in the batch-intensive workload, it does so only within the bounded variability. The predicted slopes are accurate and the major crossover point in the mixed workload is predicted accurately.

4.6 Discussion

In this chapter, we have identified and formalized five distinct data allocations which can be used to coordinate data and CPU allocations within a batch scheduling system. We have studied the influence that a range of workload and environmental characteristics can exert on these different allocations across a range of I/O intensive workloads.

Extending our formalization of the allocations, we have developed predictive models and demonstrated their accuracy in terms of selecting reasonable allocation strategies across a range of workloads and environments. We have also presented our development of a simulation framework for studying the scheduling of batch-pipeline workloads and noted the importance of visualization to validate the correctness of this simulation framework.

Several questions remain however. One of these is whether there exist additional allocation strategies beyond the five we present here. One such additional allocation would be similar to AllPrivate, but would only allocate all of the private volumes at the current depth of each pipeline and could discard earlier, no longer needed, volumes, as well as not proactively allocating volumes any earlier than is absolutely necessary. This allocation, like AllPrivate, is a strict subset of Slice, and as such, is not interesting from a scheduling perspective as whenever it is possible, then Slice will be also possible and can run at a full CPU utilization.

Although it is difficult to be certain, we are confident that if we have not identified all possible allocations, we have not missed any that are likely to outperform those that we have identified. The AllBatch, Slice, and Minimal allocations offer a balance between striving to minimize WAN utilization (as in AllBatch and Slice), and striving to minimize CPU underutilization due to barriers (as in AllBatch), and striving to maximize the CPU utilization (as in Minimal). Therefore, we believe that across the range of possible workloads, one of these allocations is likely to outperform any other.

Additional questions remain as to the validity of our assumptions concerning the scheduler's ability to garner perfect information, the homogeneity of the compute infrastructure and the canonical nature of the workloads. These questions will be addressed in more detail later in this work in Chapter 6 but it is worth a brief mention here that an understanding of canonical workload scheduling is beneficial for scheduling non-canonical workloads as well.

For example, imagine a workload with "small" batch volumes at the end and a "large" batch volume in the middle. By definition, this is not a canonical workload but by applying our scheduling knowledge we can avoid lost work that might arise otherwise. An oblivious scheduler which did not use our planning techniques but did observe capacity constraints might allow more pipelines to initially execute while the batch data is small. When the traversal would reach the large volume however, not all of the allocated pipelines would be able to remain allocated and their forward progress would be lost. This lost progress could be avoided however by making the workload appear canonical by increasing the size of all batch and private volumes to their maximum respective sizes and then using our predictive models to correctly identify the allocation limits for the workload.

In summary, this study of canonical workloads is valuable both as it provides a lexicon for discussing I/O inten-

sive workload scheduling, provides accurate predictive models for canonical batch-pipeline workload scheduling, and identifies allocation limits for non-canonical workloads. Anecdotal evidence as well as observations made in Chapter 2 suggest that many batch workloads do strictly conform to the canonical structure and only small variations in volume sizes prevent them from meeting the complete definition. All of these workloads can therefore benefit from the allocation planning provided by this study of canonical workloads.

Chapter 5

Related Work

In this chapter, we examine relevant literature and products that bear similarity to the work we have described in this dissertation. We focus our discussion around the three major parts of the dissertation. First, we examine other work done in profiling and analyzing workloads. Second, we describe related work in storage systems and distributed file systems. Finally, we look at other scheduling work from databases, parallel computing, and batch computing.

5.1 Profiling

The CPU, memory, communication, and I/O characteristics of applications have been studied for many years by the research community. These can be roughly categorized by the type of workloads that they consider: general-purpose workloads containing many applications, sequential applications examined in isolation, or parallel applications in isolation. We summarize the work in each of these categories, focusing on those that have examined file system activity.

File system activity has been examined for a range of general-purpose workloads. Many of the studies that have greatly influenced file system design over the last 20 years focused on academic and research workloads [13, 78, 89, 96]. These studies have found that most files have very short lifetimes, access patterns exhibit a high degree of locality, and read-write sharing is rare. However, missing from these broad studies of traffic is any linkage to the applications that generate the traffic.

More similar to our work are those studies that have focused on the behavior of individual applications in commercial workloads [14, 66]. However, in this domain, the interaction or pipeline behavior of sequential applications has not been examined. While we believe it may also be interesting to study the detailed memory-system behavior of our batch-pipeline workloads, we do not believe the opportunities for sharing are fundamentally different than in other studies.

Parallel applications are in many ways the most similar to pipelined batch applications. The CPU, memory, communication, and I/O behavior of parallel and vector applications have been quantified in a number of studies [31, 116, 117]; a few of which consider the impact of explicit I/O [2, 30, 90]. Our study complements these works by studying the sharing behavior of an important new class of workload.

Many of these studies demonstrate the drastic differences in I/O behavior for parallel applications compared to general-purpose workloads. For example, parallel scientific workloads often have high, bursty I/O rates [74] and

relatively constant behavior across different runs and input parameters [80]; further, parallel workloads tend to be dominated by the storage and retrieval costs of large files, particularly check-point files [74]; finally, quick deletion is uncommon [62].

5.2 Distributed File-Systems

In designing our batch-aware distributed file-system, we drew on related work from a number of distinct areas.

The manner in which the scheduler constructs private namespaces for running workloads is reminiscent of database views [55]. However, a private namespace is simpler to construct and maintain; views, in contrast, present systems with many implementation challenges, particularly when handling updates to base tables and their propagation into extant materialized views.

There has been much recent work in peer-to-peer storage systems [3, 8, 32, 61, 76, 91, 93]. Although each of these systems provides interesting solutions to the problem domain for which they are intended, each falls short when applied to the context of batch workloads, for the same reasons that distributed file systems are not a good match. However, many of the overlays developed for these environments may be useful for communication between clusters, something we plan to investigate in future work. In fact, the data distribution techniques within the BAD-FS cooperative cache are quite similar to the distributed hash tables described by Litwin *et al.* [68], and Gribble *et al.* [54].

Similar to p2p is work within grid computing [43], which uses many of the same techniques but is designed, as is BAD-FS, for c2c environments. One such example is Cluster-on-Demand [25] which offers sophisticated resource clustering techniques that could be used by BAD-FS to form cooperative cache groupings.

Also relevant within grid computing is the FreeLoader distributed storage framework [113] which builds a distributed storage system from a collection of distributed computational nodes in a manner very similar to that in our BAD-FS distributed file system. However, our focus has been on building a storage system from dedicated compute clusters while FreeLoader examines the related but more challenging question of using scavenged desktop storage resources. Previous work [20] has measured the difficulty in building storage services from a dynamically changing set of resources as would be expected in a system built from scavenged resources. However, the authors here avoid this difficulty by assuming that users of their system will continue to donate resources even when not actively using the system themselves. Given their target environment within scientific organizations, this assumption seems reasonable. A further interesting aspect of their work is their use of asymmetric striping to improve performance across the storage layer. We considered this approach initially within BAD-FS, but abandoned it for other pursuits due to the high complexity of implementation. If this approach continues to perform as expected within the FreeLoader system, we will happily re-add it to the BAD-FS design. However, although similar, the FreeLoader system cannot by itself be considered a replacement for BAD-FS as it does not provide any mechanisms, such as our lots, for exporting storage control to a higher level of software.

Extensible systems however do share our approach of allowing the application more control [19, 37, 98]. The Exokernel focus is on protection rather than policy but is motivated by the same observation that the internal system is unable to derive the necessary information itself. Although recent work has recently revisited this approach [10], extensible systems have not been commercially successful because the need for specialized policies is not so great. We believe this need is even greater for batch workloads running on systems designed for interactive use.

Some research in mobile computing bears similarity as well. Flinn *et al.* discuss the process of data staging

on untrusted surrogates [40]. In many ways, such a surrogate is similar to the BAD-FS storage server; the major difference is that the surrogate is primarily concerned with trust, whereas our servers are primarily concerned with exposing control. Both Zap [77] and VMWare [95] allow for the checkpointing and migration of either processes or operating systems. We create a remote virtual environment, but at the much higher level of a batch system. Systems with secure interposition such as Janus [51] complement BAD-FS as they should make resource owners more willing to donate their resources into shared pools.

Finally, BAD-FS is similar to other distributed file systems. The Google File System [50] was also motivated by workloads that deviate from earlier file system assumptions. An additional similarity is a simplified consistency implementation; however, GFS must relax consistency semantics to enable this, while BAD-FS does so through explicit control. Earlier work on Coda, and AFS before it, is also applicable [59]. These systems use caching for availability, so as to allow disconnected operation. In BAD-FS, storage servers enact a similar role.

5.3 Scheduling

Of course, scheduling is an old problem with relevance in many areas outside of computer science. However, we focus our attention on the related literature strictly within computer science paying particular attention to the scheduling of parallel programs, the scheduling done within database query planning, and data-driven scheduling in batch computing.

5.3.1 Parallel Scheduling

Scheduling for parallel programs in which multiple jobs communicate with each other and form a logical unit of work is very similar to our study here of batch-pipeline workloads. The difference is that the jobs within a parallel program typically communicate explicitly and during execution with each other using a networking protocol such as MPI or PVM; in contrast, jobs with a batch-pipeline workload communicate indirectly through the file system.

Backfilling techniques in parallel scheduling which use "small" programs to fill "holes" left when a program does not use all available resources [67] should be relevant for batch schedulers as well. One difference however is what constitutes a "small" program may be different; for parallel scheduling, a small program is one which uses a small number of processors whereas for data-driven batch scheduling, a small program is one which use a small amount of storage. Of course, the question of accurate runtime estimates is important for backfilling to ensure the the backfilled programs do not exceed the reservation of the originally scheduled program [39].

Perhaps the work that bears the closest similarity to data-driven batch scheduling is work in gang scheduling with memory considerations [15]. Here the authors implement a policy for capacity-aware scheduling of parallel programs where memory is the scarce resource. This is similar to our capacity-aware scheduling where our scarce resource is disk storage. One difference however is that they are studying the problem of scheduling across multiple parallel programs while we are looking at the problem of scheduling multiple jobs with a single batch-pipeline workload. Additionally, we have the luxury of refetching batch data when remote storage is scarce; they consider no such secondary backing store for memory (using virtual memory and transparently swapping to disk is a particularly severe penalty in parallel program as it interferes with the synchronization among the job's threads).

Another approach to a similar problem of coordinating allocations of storage and CPUs is shown in [100]. Here the authors propose a system in which multiple reservations are attempted and then used only if all required reservations are successful. In fact, this is the approach taken by our modified batch scheduler which only executes

jobs after both claiming CPUs from a match-making system and successfully allocating storage from the batch-aware distributed file system.

5.3.2 Database Query Planning

In a manner similar to that of a batch-pipeline scheduler, query optimizers in many relational database management systems (*DBMS*) create execution plans [84]. Like a batch-pipeline scheduler, queury optimizers attempt to reduce total completion time for their workloads. These workloads are described by the user in a query language such as SQL. Even more so than batch-pipeline workload descriptions, these query languages are relational and describe to the DBMS *what* work needs to be done and do not contain directions for *how* the work should be done.

Query optimizers rely on detailed cost estimates in order to select the "best" query execution plan (*QEP*) for any given query. These cost estimates are dependent upon estimated values for the number of rows that will appear in temporary tables or *materialization points* within the many stages of complex queries.

Like a batch-pipeline scheduler, query optimizers also use both environmental and workload information in order to determine the best QEP. However query optimizers must consider a third type of information which does not exist in a batch-pipeline scheduling system: information about the database itself. One reason for this third information source is that the database in a DBMS is structurally identical to the batch data in a batch-pipeline workload. A batch-pipeline scheduler must create execution plans for workloads over disjoint batch data sets and thus each data set must be described in the particular workload; conversely a query optimizer is uniquely bound to one database and therefore queries need not specify anything about it. This information about the database however is used by query optimizers in much the same way that a batch-pipeline scheduler uses information about batch data.

In fact, the execution of many queries can even more closely resemble a batch-pipeline pipeline in that it each is uniquely defined by endpoint inputs (the original query) and endpoint outputs (the results) and may produce and consume temporary data (intermediate results such as temporary tables or materialization points). As such, query optimizers must use predictions about the size of the data (*i.e.* their cardinality) in order to make estimates about the cost of a query plan. These estimates are then combined with the environmental information to inform the cost estimates. This environmental information is the same as that used by a batch-pipeline scheduler consisting of various estimates for the physical capacities and bandwidths of the computational resources.

One key difference however is that query optimizers have the flexibility to devise multiple orderings for their query by, for example, rearranging the orders of selects and joins. Although a batch-pipeline scheduler does not have quite this flexibility in terms or rearranging orderings, it can select different traversal patterns such as those dictated by the AllBatch, Slice, and Minimal data allocations.

Inaccurate information

Query optimizers are similarly dependent on accurate information in order to make informed plans. However, inaccurate information and predictions as to the likely selectivity of a particular table for example does not create the possibility of *correctness* problems as inaccurate information can do in a batch-pipeline scheduler but only affects the ability of the query optimizer to improve the runtime performance of the query execution plan.

To mitigate the effects of inaccurate and imprecise information, IBM has developed LEO [73], a learning optimizer. This learning optimizer bears great similarity to my dissertation as it monitors predictions and constantly

refines them as they prove to be inaccurate. Subsequent queries then use the more accurate estimates. Proactive reoptimization [12] extends this idea by dynamically using refreshed information to change the current QEP. These systems show the potential to reduce or even eliminate the need for costly off-line statistic gathering sessions and offer promise that similar information-gathering techniques within a batch-pipeline scheduler could reduce the need for the user to provide accurate and detailed workload estimates.

5.3.3 Data-Driven Batch Scheduling

Workflow management has historically been the concern of high-level business management problems involving multiple authorities and computer systems in large organizations, such as approval of loans by a bank or customer service actions by a phone company [49]. Our scheduler works at a lower semantic level than such systems; however, it does borrow several lessons from them, such as the integration of procedural and data elements [92]. The automatic management of dependencies for both performance and fault tolerance is found in a variety of tools [21].

Many other systems have also managed dependencies among jobs. A basic example is found with the UNIX tool `make`. More elaborate dependency tracking has been explored in Vahdat and Anderson's work on transparent result caching [110]; in that work, the authors build a tool that tracks process lineage and file dependency automatically. Our workflow description is a static encoding of such knowledge.

BAD-FS could be further improved through the prefetching of batch datasets. Other work [24] has noted the difficulty in correctly predicting future access patterns. In BAD-FS, however, these are explicitly supplied by the user via the declarative workflow description.

Within the grid community, there is an increasing awareness of the growth of datasets [8, 43, 53, 56, 113] and, not surprisingly, a corresponding increasing interest in the coordinated scheduling of data and computation. Stork [60] creates mechanisms for the controlled transfer of datasets across wide-area networks and, like our system, is explicitly designed for moving an awareness of batch inputs and endpoint outputs into the scheduling framework. However, Stork is a procedural approach where the user adds explicit data-movement jobs into their workloads whereas we have opted for a declarative approach in which the user provides the necessary information to the scheduler which then proceeds as it deems best. Our current implementation fetches batch data on demand whereas Stork necessarily pre-fetches the data before executing the jobs. We note however that pre-fetching in this manner is allowed within our framework and Stork would be ideal tool to accomplish this. Further, Stork provides no mechanism for localizing temporary pipeline.

Many approaches have been proposed for caching and locating batch datasets [16, 17, 26, 63, 79, 86, 88]. These approaches complement our work here as they pertain to the more persistent question of what happens to batch data *following* the completion of the workload, whereas here we have addressed the question of how to access the batch data *during* the workload's execution.

Chapter 6

Conclusions

Large-scale modern batch scheduling systems must be data-aware. So long as batch jobs are compute-bound then existing batch scheduling systems are sufficient. However, as we have seen here, there exists a large class of important scientific and industrial batch workloads that perform large amounts of data access and are I/O bound rather than compute bound. Recent trends show that the increase in application data size is increasing faster than the increase in computational power [53], *i.e.* applications are becoming increasingly data-intensive.

We profile several of these workloads and create a taxonomy by which we can describe them. Further, we demonstrate the need for distributed file system support for efficient scheduling of these workloads and show the value of this in one such implementation.

We define several scheduling allocations for data-intensive batch workloads and evaluate these across a range of workload and environmental characteristics. Finally, we provide an analytical predictive model with which a data-aware batch scheduler can choose the appropriate data allocation with a goal towards minimizing the total time to completion.

In this chapter, we'll first summarize our major contributions and then speculate on new areas of research that are now possible due to this work as well as areas that were previously possible but may now be more interesting in light of our work here and therefore may warrant re-examination. We then conclude the dissertation.

6.1 Summary

6.1.1 Profiling

The analysis in Chapter 2 of data-intensive scientific batch workloads provides the working foundation for this entire dissertation. The motivation for building data-aware batch schedulers is derived from this data analysis as we quantified the exact degree of data-intensity in each applications which had previously been characterized as data-intensive in anecdote only.

The first contribution of the workload profiling is to provide this three way differentiation of I/O types between batch data, pipeline data, and endpoint data. This differentiation is the enabling factor that allows informed coordinated coallocation of data and CPUs by data-aware batch schedulers.

With these measurements of I/O intensity, we define scaling limitations for our studied workloads. Given reasonably sized datasets and compute environments, we show how failing to differentiate between batch data, pipeline data, and endpoint data can quickly overwhelm current storage systems.

6.1.2 File System Support

The prototype data-aware batch system in Chapter 3 makes three important contributions to batch scheduling. First, we have shown how batch schedulers can use information about I/O types and quantities to plan a workflow traversal that avoids storage overallocations and makes informed and dynamic data replication decisions. Second, we have discussed the feasibility of current distributed file systems and found them wanting. Finally, we have designed a new batch-aware file system and used a prototype implementation along with our modified batch scheduler to show runtime improvements of several orders of magnitude.

Data-aware schedulers

By assuming that users can provide accurate and basic information about the I/O behavior in their workloads, we have designed, and implemented, a modified batch scheduler system based on the current Condor system. This new data-aware Condor scheduler carefully plans data allocation using a technique of scoping the workload I/O.

By caching batch data and localizing pipeline data, the scheduler minimizes the amount of data crossing the wide-area bottleneck connection between home storage servers and compute clusters. The scheduler also uses this information to make careful job placement decisions ensuring that the output of running jobs will not overallocate available storage. Finally, using dynamically observed values for the cost of initial data replication, the cost of subsequent data replication, and failure rates, we have designed and implemented a scheduler which makes *individualized* data replication decisions. These scheduling enhancements are made possible by our taxonomy of data types, our assumption that users can provide information, and through the exposure of explicit storage control in our new batch-aware distributed file system.

Batch-aware distributed file systems

As we demonstrated in Chapter 3, current existing distributed file systems such as AFS and NFS are not well suited for remote execution of batch workloads. Although they are explicity designed for remote (as well as concurrent) data access, they are designed for the generalized access patterns of interactive workloads. With the workload specific information available within a batch workload, a data-aware batch scheduler can make much more informed decisions individualized to the specific batch workload.

As an example, consider the write behavior of both AFS and NFS. Writes in AFS are cached locally and flushed to the home storage server when the file is closed; in NFS, writes are flushed at some periodic interval (generally set at thirty seconds). Although both of these behaviors make sense in some situations, neither is “always” best. Conversely, with information about data types, the data-aware batch scheduler can carefully control these writes. For example, the scheduler ensures that these remote flushes never occur for temporary pipeline data and can further ensure that endpoint output data is written only once.

The main contribution of our distributed file system work is the recognition that control of storage decisions be moved from the storage elements themselves to an external controller, in this case a data-aware batch scheduler. We enable this external explicit controller through a storage abstraction called lots. Lots provide guaranteed storage allocations that allow the scheduler to make job placements with confidence that the necessary job data will be accessible.

6.1.3 Data-Driven Scheduling

A simulation and workload framework

In Chapter 4, we use simulation in order to examine fundamental algorithmic details in a highly controlled and simplified environment. We further control the problem space by simplifying our workload model, reducing the number of variables needed to define workloads, and combining these simplifications into an abstraction we term canonical workloads.

Data-driven scheduling policies

With this controlled simulation testbed and this controlled workload abstraction, we examine the allocation choices available to the batch scheduler for scheduling workloads with both job and data constraints. We run experiments across a wide range of workload and environmental characteristics to explore this scheduling space.

The main contribution of our scheduling work is the creation of five distinct data-driven scheduling policies and the identification of the performance problems each seeks to avoid. We identify that there are three possible performance problems when scheduling batch-pipeline workloads in environments in which storage is constrained. One, batch data might need to be refetched when it could otherwise be cached thereby incurring an redundate use of a potential bottleneck resource in the wide-area network connection between the storage server and the compute cluster. Two, the CPUs on the compute cluster might be underutilized due to concurrency limits imposed by the data dependencies of the workloads. Three, the CPUs might also be underutilized due to barriers within the workloads.

Each of our five scheduling policies seeks to avoid a particular performance problem possibly at the expense of the other two. The relative costs of these performance problems varies depending on specific characteristics of the workload and the compute environment. For example, the relative penalty to refetch batch data increases as the bandwidth to the storage server decreases.

Predictive modelling

Using different possible data scheduling allocations, we show there exists a wide range of possible performance and that the scheduler decision is fraught with difficulty: a bad decision can lead to a throughput loss approaching one-hundred percent. Therefore the final contribution of our scheduling work is to define an simple analytical predictive model that predicts the runtimes for scheduling workloads using each of our five scheduling policies. We demonstrate that a scheduler using our predictive model to select a scheduling policy in most cases performs within 5% of an ideal scheduler that uses a theoretical perfect predictive model and in no cases exceeds 30% percent.

6.2 Reflections

We now offer some reflections about the research process and offer advice to others pursuing similar studies.

6.2.1 Methodology

At the methodological level, we have observed the value of studying both simulations and a prototype implementation, which was also cited as valuable in both [9] and [11]. Having a highly controlled and simplified simulation allowed us to focus on fundamental algorithmic details which are sometimes obscured in the complexities and distractions of a full prototype implementation.

Additionally, the simulation environment drastically simplified our data collection as scaling to large numbers of machines was as easy as changing a parameter in the simulation and as difficult as contending for scarce resources in the implementation. The runtime performance of doing experiments was also drastically reduced; experiments which took days on the full implementation could be simulated in hours. Of course, we would not have been able to have confidence in our simulated results without the deep understanding of the full system that came from designing and managing the full implementation. Validating results that show similar measurements from each added to this confidence.

Also of value was the predictive analytical model. In addition to being an important contribution to batch scheduling as it allows the scheduler to quickly make informed allocation decisions, the model adds further confidence in both the simulation and the prototype implementation.

6.2.2 Research Cycle

In this work, we have followed what we consider to be an ideal model for systems research: measure, build, evaluate, and refine. It is difficult, if not impossible, to propose a new system without first measuring some key aspect of the previous. We have done that here by building a foundation for our work from our initial measurement study of data-intensive batch workloads. Having this body of empirical measurement gave us insight into the problem area and helped us quantitatively define the scope of the problem so that we could then design a solution. A less technical additional advantage of proposing our new approach only *after* collecting our empirical measurements is that this empirical workload profiling study gave us credibility within the community that was instrumental in selling our new file system design.

Finally, it was only through building and then evaluating the file system design, that we made the realization that scheduling I/O intensive batch workloads was more complex than merely choosing between breadth-first and depth-first allocations. This realization then led to our work on scheduling allocations and our predictive analytical models.

6.3 Future Work

Though we have made significant inroads both in designing file system support for data-intensive batch workloads and in designing data-aware batch schedulers, much work remains to be done. We now briefly address some key areas that are not addressed elsewhere in this dissertation. We note further that much of this additional work was not strictly possible before; these new areas of exploration are now possible because of the progress we have described here.

6.3.1 More measurement

The main limitation of our workload analysis in Chapter 2 is the breadth of the study. While instrumental in defining the different types of I/O in batch workloads and revealing that several important scientific applications are heavily I/O bound, this analysis would benefit from additional study.

Several interesting questions specifically related to batch-pipeline workload scheduling remain. One thing unmeasured in our initial study is variability across multiple instances of the same workload. We studied the effect of runtime variability in Chapter 4, but did so without any empirical measurements to guide us. It would be interesting to determine just how variable are batch submissions. This variability will directly effect scheduling decisions; the less variable, the easier the planning.

Additional questions that would be interesting to study are whether non-scientific workloads display similar patterns of I/O behavior (*i.e.* are they batch-pipeline in structure). Although there is no reason to suspect otherwise, validating this would be useful.

Further, it would be interesting to know what percentage of all batch jobs are data-intensive. Of course, defining exactly what data-intensive means would be necessary here as well. For this thesis, we defined data-intensive to mean two things: one, that the workload has data constraints in that the total amount of data does not fit within available storage and two, that the workload spends at least fifty percent of its runtime performing I/O when executing in a remote environment using remote I/O.

Finally, information that is vital towards knowing whether our analytical predictive model is immediately useful in batch scheduling or must be further refined is to quantify what percent of current batch workloads are canonical and to measure to what degree do non-canonical workloads violate the canonical definition. These measurements may suggest that additional work is required to study the scheduling of non-canonical workloads.

6.3.2 Non-canonical Workloads

Should non-canonical workloads prove prevalent, additional study of scheduling these workloads should prove fruitful. There are several ways in which workloads can be non-canonical and each of these should provide different scheduling challenges. Workloads which do not fit the standard batch-pipeline structure (*e.g.* a parent job having multiple children) do not fit within our predictive analytical model nor do they easily fit within our different scheduling allocations. These workloads, which have been examined in the simpler case in which data is not constrained as in the DAGMan scheduler [29], should prove significantly more challenging when they impose additional data constraints.

Workloads which are not strictly canonical due to volume sizes should fit much more easily within our scheduling framework. Indeed, our scheduling framework is still extremely useful for these types of workloads because it can be used to provide limits on execution. For example, imagine a workload in which a batch volume at a depth d is sufficiently large that while it is allocated, only p pipelines can execute concurrently. If batch volumes at shallower depths are smaller than this batch volume at depth d, an uninformed scheduler might mistakenly begin executing more than p pipelines. Only after advancing the workflow to depth d, will the scheduler realize the error of its way and be forced to abandon (and lose the progress) in all but p pipelines. Using our scheduling framework however, the scheduler will plan for this bottleneck at depth d and ensure that no more than p pipelines execute concurrently.

However, this slack at depths less than d is itself interesting and worthy of investigation. It may be that

other areas of the same workload might fit within this slack. Additionally, in multi-user and multi-workload environments, this slack may prove useful.

6.3.3 Multi-* Effects

In this thesis, we have studied extensively the base case of a single user running a single workload on a single compute cluster. Extending our scheduling framework to multiple users, multiple workloads, and multiple clusters is a challenge for future study.

Multi-user and multi-workload

The challenges for multiple users and multiple workloads seem essentially identical. Each distinct user provides distinct workloads and therefore the multi-user problem essentially becomes the multi-workload problem.

However, there does exist at least one distinction. Users in batch systems control the relative priority of their own workloads. The relative priority of workloads owned by different users is subject to a fairness policy defined by the batch scheduler. Although much work has been done in multi-user computing both in time-sharing interactive and batch systems, the additional complexities of the dual scheduling concerns of CPU and data allocation make a re-examination of multi-user scheduling necessary for data-intensive batch workloads.

For example, existing sharing policies that split resources in space (*i.e.* they give a fraction of a compute cluster to each user) need to be aware of the data constraints of the workload. If some workload w needs sixty percent of the available storage in order to allow a efficient data allocation schedule, then an uninformed scheduler which splits CPU's evenly across users will not provide workload w an effective compute platform. Related scheduling work in parallel program scheduling should provide inspiration here.

However, we note that a defining aspect of the success of the Condor batch scheduler has been the mantra of its founder Miron Livny who scorns optimization and prefers systems which "just work." Embracing this philosophy with regards to the scheduling of multiple batch-pipeline workloads is perhaps the best initial approach here. A simple solution which gave entire compute clusters to a workload until the completion of that workload might prove just as worthwhile in terms of long term throughput measurements as a more complex solution which attempts to find optimal best fits. We note that scheduling multiple workloads each of which require some percentage of resource is effectively a bin-packing problem which is known to be NP-complete. Further any computational slack within these workloads due to data constraint can be easily filled with CPU-intensive jobs.

Multi-cluster

Additional complexity occurs when considering how to place workloads with data constraints across multiple clusters. This problem seems strikingly similar to memory allocation in operating system kernels. Any placement is likely to lead to fragmentation of these compute clusters. The various memory allocation policies such as best-fit, worst-fit, and first-fit are a likely good initial starting point for an examination of scheduling data constrained workloads across multiple clusters of compute resources.

One additional aspect that might prove challenging is when a particular workload has a possible data allocation that exceeds the available storage of any one compute cluster. In such a case, combining multiple clusters might prove worthwhile although we note that to do so efficiently will require additional changes in both the design and the implementation of our prototype batch-aware distributed file system.

Here again the problem quickly becomes NP-complete as the scheduler attempts to evaluate across an infinite number of possible allocations and combinations and fragmentations. Perhaps an initial approach to this problem should strive to merely design a system that "just works" and then seek to improve upon that.

6.3.4 Inaccurate Information

A main limitation of our scheduling work in both Chapters 3 and 4 is our assumption that perfect and complete information about the environment and the workload can be provided to the scheduler. While we believe that the information provided about the environment by tools such as the Network Weather Service [115] should be sufficiently accurate to not adversely effect our scheduling decisions, we have no such confidence in the users' ability to provide accurate workload information. As users have been historically observed to have difficulty providing accurate runtime estimates [27, 65, 75], it is unlikely that they can provide accurate I/O estimates.

How batch schedulers can maintain levels of throughput while basing their allocation decisions on inaccurate workload information is a challenging problem we leave for future researchers. We are gratified to see that others are examining the challenge of scheduling with inaccuracies, albeit in more standard batch workloads without data constraints [82].

Additional recent work in databases also examines the thorny question of scheduling without accurate information. Query optimizers are similarly dependent on accurate information in order to make informed plans. To mitigate the effects of inaccurate and imprecise information, IBM has developed LEO [73], a learning optimizer. Subsequent queries then use the more accurate estimates. In fact, LEO shows the potential to reduce or even eliminate the need for costly off-line statistic gathering sessions. This suggests that similar information-gathering techniques within a batch-pipeline scheduler could reduce or eliminate the need for the user to provide accurate workload estimates.

Another possible source for inspiration in this area of scheduling with inaccurate information might be gleaned from similar problems in computer architecture. For example, branch prediction in a processor core is an attempt to make a schedule with imperfect information.

Information gathering

Another approach to this same challenge would be attempting to find alternative sources beyond the user for workload information. Should workloads prove to be highly uniform, using historical information may be useful here. The challenge here then becomes one of information storage and retrieval as well as defining exactly what information should be stored.

Finally, batch schedulers could attempt to gather information dynamically from running workloads. Again, decisions made using this dynamically gathered information need to be influenced by the degree of variability within the workload. Gathering the variability information dynamically should also be possible but a larger sample size may be needed.

We note here that the question of inaccurate information is significantly more complex in the context of non-canonical workloads than it is for canonical workloads. Gathering accurate information about a canonical workload is significantly easier that gathering information about a non-canonical one as each pipeline in a canonical workload is essestially identical (with the exception of runtime variability). To schedule canonical workloads without accurate information is a straightforward challenge; all the scheduler must do is execute some number of pipelines

initially and then use measured observations about those initial pipelines to select an allocation plan for the remaining.

In spirit, this seems similar to a technique used by Google for dealing with the possibility of queries which cause their compute machines to crash [7]; such queries are referred to as "queries of death." These queries of death refer to queries executed in parallel across a cluster of machines which cause all these machines to crash; to prevent this large failure across the cluster, an individual "probe" query is run; if successful, the entire parallel query is performed; otherwise, it is not. The total expected slowdown is less than one hundred percent because a single query is expected to run at the average speed while the entire parallel query runs at the speed of the slowest node.

Similar probing could be done by batch schedulers, not to avoid massive correlated cluster failure, but to collect the information needed for planned data allocation. In such a case, the slowdown could be even less than that in the Google case due to the relative widths of the workload and the cluster. For Google, a parallel query is run only once across the full cluster whereas batch workloads wider than the number of available compute nodes will run for multiple iterations thereby amortizing the slowdown effect imposed by the probe pipeline.

6.3.5 Dynamic Reallocation

In Chapter 4 we show how our predictive analytical model can be used to select between different data scheduling allocations. The relative performance of the different allocations depends on various characteristics of the workload and the environment.

We note here that some of these characteristics change during the course of the execution of the workload. For example, for each completed pipeline, the width of the workload is reduced by one. It should therefore be possible to dynamically re-evaluate the remaining workload and switch to a different allocation if the model predicts that doing so will reduce the remaining total time to completion. Challenges here include defining the frequency of the re-evaluation and actually switching between different allocations.

In some instances it should be trivial to switch allocations; for example there are no limitations involved in switching from a Slice allocation to a Minimal one. However, other transistions are more involved. For example, switching from an AllBatch allocation to a Slice allocation will require removing some of the batch volumes. If currently executing pipelines are accessing those volumes, the cost of removing them needs to be weighed against the benefit of switching allocaitons.

This periodic dynamic evaluation of the data allocation plan bears similarity to recent work in proactive reoptimization in database queries [12]. As in database queries, estimates about the workload (or query) information can be continuously updated during the traversal of the workflow (execution of the query). Due to this similarity, many of the techniques developed there may be applicable here. In particular, the use of *bounding boxes* to represent uncertainty in statistics should be useful here as well. A batch scheduler could use bounding boxes to represent the complete set of allocations that are possible within some degree of error in the workload information and can continuously evaluate only these allocations as opposed to the complete set. However, we note that this may be somewhat less useful here as the set of possible query plans is potentially infinite whereas the complete set of possible scheduling allocations is only three.

6.3.6 Checkpointing

In Chapter 3 we examine a cost-benefit model that can make individualized replication decisions for each output pipeline volume in a workload. Replicating these volumes effectively checkpoints the pipeline at the depth at which it wrote that volume. One limitation here is that we did not consider the storage implications of replicating these volumes.

Our cost-benefit replication model assumes that storage is not constrained and thus the sole cost it considers for replicating data is the cost to replicate. For data constrained workloads, there is an additional cost to replicate data: doing so further constrains the workload and may make some allocation schedules no longer possible or could reduce the maximum concurrency of the workload.

We note further that some systems, such as the Condor batch scheduling system, checkpoint the process image to an external checkpoint server. Clearly checkpointing only the process for a job in a batch-pipeline workload is insufficient as these jobs have additional data requirements. Therefore, additional work remains to combine the checkpointing of processes with the replication of their volumes.

6.3.7 Partial Results

One simplifying assumption that we make in our work is that users are satisfied with receiving the results of their workloads all at once; in other words that users are not interested in partial results. However, there do exist many situations in which this is not the case.

There has been recent work in this area [70, 97] to help users dictate the order in which their pipelines are executed and to provide these users with partial results as individual pipelines are executed. Additionally, these systems then allow users to steer the workloads towards more interesting areas within the parameter space.

For example, imagine a user who wants to run a parameter sweep across some variable v for each possible value of v between 0 and 1000. The user suspects that there is an interesting area within this parameter sweep but does not know where it occurs. In such a case the scheduler can first run pipelines at a coarse granularity of v, for example for 0, 100, 200, etc. Seeing partial results, the user may notice an unexpected discrepancy between the values at say 300 and 400 and can then direct the scheduler to focus on this area.

These partial results systems assume the absence of data constraints. Adding this additional constraint to the scheduling decisions is a challenging problem we leave for future study.

6.4 Postscript

Given the continued emergence of increased wide-area colloborations (as attested by the media attention [28, 71, 72, 105] surrounding grid computing [45]) and the continued predictions of increasingly large data-sets such as those made about the ATLAS and CMS physics projects [43] and astrophysics data [53], batch scheduling needs new models for scheduling data constrained workloads in remote environments.

Existing batch schedulers plan for CPU allocation only; data movement happens as a side-effect of job placement. For CPU intensive jobs in a local environment, this is a perfectly reasonable trade-off. Any penalty incurred for not planning the data movement is negligible and well worth the simplicity of the remote I/O model of direct access to remote data. However, this penalty grows quickly as the the data sets grow in both size and in distance.

These data constrained workloads do have well defined patterns of I/O behavior. By leveraging information about these workloads, batch schedulers can plan for coordinated data and CPU allocations. However, in order to do so, they need complicity from the distributed file system. This concept of external explicit storage control is one of the core contributions of this dissertation.

We have found that two mechanisms provide what is needed here. First, guaranteed storage allocations within the distributed file system allow the external scheduler to make appropriate storage decisions concerning data caching, replication, and consistency. Second, our predictive analytical model allows the scheduler to make the appropriate storage decisions.

Allocating CPUs is easy – but may cause an overallocation of storage; allocating data is also easy – but may cause an underutilization of CPU; allocating both so that neither is adversely affected is the challenge for modern batch scheduling systems.

Bibliography

[1] LHC - The Large Haldron Collider Home Page. `http://lhc-new-homepage.web.cern.ch/lhc-new-homepage/`, 2004.

[2] Anurag Acharya, Mustafa Uysal, Robert Bennett, Assaf Mendelson, Michael Beynon, Jeffrey K. Hollingsworth, Joel Saltz, and Alan Sussman. Tuning the performance of I/O intensive parallel applications. In *Proceedings of the Fourth Workshop on Input/Output in Parallel and Distributed Systems (IOPADS)*, pages 15–27, Philadelphia, Pennsylvania, 1996.

[3] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[4] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Database Mining: A Performance Perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914–925, December 1993.

[5] Stephen F. Altschul, Thomas L. Madden, Alejandro A. Schaffer, Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. In *Nucleic Acids Research*, pages 3389–3402, 1997.

[6] Gene Amdahl. Storage and I/O Parameters and System Potential. In *IEEE Computer Group Conference*, pages 371–72, June 1970.

[7] Darrell Anderson. *Personal Communication*, 2004.

[8] T.E. Anderson, M.D. Dahlin, J.M. Neefe, D.A. Patterson, and R.Y. Wang. Serverless Network File Systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 109–26, Copper Mountain Resort, Colorado, December 1995.

[9] Andrea C. Arpaci-Dusseau. *Implicit Coscheduling: Coordinated Scheduling with Implicit Information in Distributed Systems*. PhD thesis, University of California, Berkeley, 1998.

[10] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Nathan C. Burnett, Timothy E. Denehy, Thomas J. Engle, Haryadi S. Gunawi, James Nugent, and Florentina I. Popovici. Transforming Policies into Mechanisms with Infokernel. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 90–105, Bolton Landing (Lake George), New York, October 2003.

[11] Remzi H. Arpaci-Dusseau. *Performance Availability for Networks of Workstations*. PhD thesis, University of California, Berkeley, 1999.

[12] Shivnath Babu, Pedro Bizarro, and David DeWitt. Proactive Re-Optimization. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD '05)*, Baltimore, Maryland, June 2005.

[13] Mary Baker, John Hartman, Martin Kupfer, Ken Shirriff, and John Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 198–212, Pacific Grove, California, October 1991.

[14] Luiz Andre Barroso, Kourosh Gharachorloo, and Edouard Bugnion. Memory system characterization of commercial workloads. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 3–14, 1998.

[15] Anat Batat. Gang scheduling with memory considerations. In *IPDPS '00: Proceedings of the 14th International Symposium on Parallel and Distributed Processing*, page 109, Washington, DC, USA, 2000. IEEE Computer Society.

[16] W. Bell, D. Cameron, R. Carvajal-Schiaffino, A. Millar, K. Stockinger, and F. Zini. Evaluation of an Economy-Based File Replication Strategy for a Data Grid. In *In Proceedings of 3nd IEEE Int. Symposium on Cluster Computing and the Grid (CCGrid'2003)*, Tokyo, Japan, May 2003.

[17] John Bent, Doron Rotem, Alexandru Romosan, and Arie Shoshani. Coordination of Data Movement with Computation Scheduling on a Cluster. In *Challenges of Large Applications in Distributed Environments (CLADE 2005)*, Research Triangle Park, North Carolina, July 2005.

[18] John Bent, Venkateshwaran Venkataramani, Nick Leroy, Alain Roy, Joseph Stanley, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Miron Livny. Flexibility, Manageability, and Performance in a Grid Storage Appliance. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC 11)*, pages 3–12, Edinburgh, Scotland, July 2002.

[19] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 267–284, Copper Mountain Resort, Colorado, December 1995.

[20] Charles Blake and Rodrigo Rodrigues. High Availability, Scalable Storage, Dynamic Peer Networks: Pick Two. In *The Ninth Workshop on Hot Topics in Operating Systems (HotOS IX)*, Lihue, Hawaii, May 2003.

[21] Yuri Breitbart, Andrew Deacon, Hans-Jorg Schek, Amit P. Sheth, and Gerhard Weikum. Merging Application-centric and Data-centric Approaches to Support Transaction-oriented Multi-system Workflows. *SIGMOD Record*, 22(3):23–30, 1993.

[22] Jason F. Cantin and Mark D. Hill. Cache Performance for Selected SPEC CPU2000 Benchmarks. Computer Architecture News (CAN), September 2001.

[23] Satish Chandra, Michael Dahlin, Bradley Richards, Randolph Y. Wang, Thomas E. Anderson, and James R. Larus. Experience with a Language for Writing Coherence Protocols. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, Santa Barbara, California, October 1997.

[24] Fay W. Chang and Garth A. Gibson. Automatic I/O Hint Generation Through Speculative Execution. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 1–14, New Orleans, Louisiana, February 1999.

[25] Jeffrey S. Chase, Laura E. Grit, David E. Irwin, Justin D. Moore, and Sara Sprenkle. Dynamic Virtual Clusters in a Grid Site Manager. In *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC 12)*, Seattle, Washington, June 2003.

[26] Ann L. Chervenak, Ewa Deelman, Ian Foster, Adriana Iamnitchi, Carl Kesselman, Wolfgang Hoschek, Peter Kunst, Matei Ripeanu, Bob Schwartzkopf, Heinz Stockinger, Kurt Stockinger, and Brian Tierney. Giggle: A framework for constructing scalable replica location services. In *Supercomputing 2002*, Baltimore, Maryland, November 2002.

[27] Su-Hui Chiang, Andrea Arpaci-Dusseau, and Mary K. Vernon. The Impact of More Accurate Requested Runtimes on Production Job Scheduling Performance. In *8th Workshop on Job Scheduling Strategies for Parallel Processing*, Edinburgh, Scotland, July 2002.

[28] Don Clark. Sun Expands its Plan to Market Utility-Style Computer Services. *The Wall Street Journal*, February 1, 2005.

[29] Condor. The Condor Directed-Acyclic-Graph Manager (DAGMan). http://www.cs.wisc.edu/condor/dagman/, 2002.

[30] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of the IEEE/ACM Conference on Supercomputing*, San Diego, California, 1995.

[31] R. Cypher, A. Ho, S. Konstantinidou, and P. Messina. Architectural Requirements of Parallel Scientific Applications with Explicit Communication. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)*, San Diego, California, May 1993.

[32] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-Area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.

[33] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, Monterey, California, November 1994.

[34] P. J. Denning. The Working Set Model for Program Behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[35] EDA Industry Working Group. The EDA Resource. http://www.eda.org/, 2003.

[36] David A. Edwards and Martin S. McKendry. Exploiting Read-Mostly Workloads in The FileNet File System. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP '89)*, pages 58–70, Litchfield Park, Arizona, December 1989.

[37] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain Resort, Colorado, December 1995.

[38] Michael J. Feeley, W. E. Morgan, F. H. Pighin, Anna R. Karlin, and Henry M. Levy. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 201–212, Copper Mountain Resort, Colorado, December 1995.

[39] Dror G. Feitelson, Larry Rudolph, and Uwe Schwiegelshohn. Parallel Job Scheduling — A Status Report. In *Job Scheduling Strategies for Parallel Processing*, New York, New York, June 2004.

[40] Jason Flinn, Shafeeq Sinnamohideen, Niraj Tolia, and M. Satyanarayanan. Data Staging on Untrusted Surrogates. In *Proceedings of the 2nd USENIX Symposium on File and Storage Technologies (FAST '03)*, San Francisco, California, April 2003.

[41] J.A. Foley. An integrated biosphere model of land surface processes, terrestrial carbon balance, and vegetation dynamics. *Global Biogeochemical Cycles*, 10(4):603–628, 1996.

[42] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullman, Godmar Back, and Steven Clawson. Microkernels Meet Recursive Virtual Machines. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, Seattle, Washington, October 1996.

[43] Ian Foster and Paul Avery. Petascale Virtual Data Grids for Data Intensive Science. GriPhyn White Paper, 2001.

[44] Ian Foster, Carl Kesselman, Gene Tsudik, and Steven Tuecke. A Security Architecture for Computational Grids. In *Proceedings of the 5th ACM Conference on Computer and Communications Security Conference*, pages 83–92, 1998.

[45] Ian Foster, Carl Kesselman, and Steve Tuecke. The Anatomy of the Grid: Enabling Scalable Virtual Organizations. *International Journal of Supercomputer Applications*, 15(3), 2001.

[46] Ian Foster, Jens Voeckler, Mike Wilde, and Yong Zhou. Chimera: A virtual data system for representing, querying, and automating data derivation. In *Proceedings of the 14th Conference on Scientific and Statistical Database Management*, Edinburgh, Scotland, July 2002.

[47] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steven Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC 10)*, San Francisco, California, August 2001.

[48] Erol Gelenbe. On the Optimal Checkpoint Interval. *Journal of the ACM*, 26(2):259–270, Apr 1979.

[49] Dimitrios Georgakopoulos, Mark F. Hornick, and Amit P. Sheth. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases*, 3(2):119–153, 1995.

[50] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 29–43, Bolton Landing (Lake George), New York, October 2003.

[51] Ian Goldberg, David Wagner, Randy Thomas, and Eric Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the Sixth USENIX Security Symposium*, July 1996.

[52] Jim Gray and Prashant Shenoy. Rules of thumb in data engineering. In *Proceedings of the Sixteenth IEEE International Conference on Data Engineering (ICDE)*, pages 3–12, 2000.

[53] Jim Gray and Alexander S. Szalay. Scientific Data Federation: The World-Wide Telescope. In Ian Foster and Carl Kesselman, editors, *The Grid 2: Blueprint for a New Computing Infrastructure*, pages 95–108. 2003.

[54] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, San Diego, California, October 2000.

[55] Ashish Gupta and Inderpal Singh Mumick. Maintenance of Materialized Views: Problems, Techniques and Applications. *IEEE Quarterly Bulletin on Data Engineering; Special Issue on Materialized Views and Data Warehousing*, 18(2):3–18, 1995.

[56] Koen Holtman. CMS data grid system overview and requirements. CMS Note 2001/037, CERN, July 2001.

[57] P.O. Hulith. The AMANDA experiment. In *Proceedings of the XVII International Conference on Neutrino Physics and Astrophysics*, Helsinki, Finland, June 1996.

[58] Michael B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP '93)*, pages 80–93, Asheville, North Carolina, December 1993.

[59] J.J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[60] Tevfik Kosar and Miron Livny. Stork: Making Data Placement a First Class Citizen in the Grid. In *Proceedings of 24th IEEE International Conference on Distributed Computing Systems (ICDCS 2004)*, Tokyo, Japan, 2004.

[61] John Kubiatowicz, David Bindel, Patrick Eaton, Yan Chen, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Westley Weimer, Chris Wells, Hakim Weatherspoon, and Ben Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 190–201, Cambridge, Massachusetts, November 2000.

[62] S. Kuo, M. Winslett, Y. Cho, J. Lee, and Y.Chen. Efficient input and output for scientific simulations. In *Proceedings of I/O in Parallel and Distributed Systems (IOPADS)*, pages 33–44, 1999.

[63] Houda Lamehamedi, Zujun Shentu, Boleslaw K. Szymanski, and Ewa Deelman. Simulation of Dynamic Data Replication Strategies in Data Grids. In *International Parallel and Distributed Processing Symposium*, Nice, France, April 2003.

[64] Tal L. Lancaster. The Renderman Web Site. http://www.renderman.org/, 2002.

[65] Cynthia B. Lee, Yael Schwartzman, Jennifer Hardy, and Allan Snavely. Are user runtime estimates inherently inaccurate. In *10th Workshop on Job Scheduling Strategies for Parallel Processing*, New York, NY, June 2004.

[66] Dennis C. Lee, Patrick J. Crowley, Jean-Loup Bear, Thomas E. Anderson, and Brian N. Bershad. Execution characteristics of desktop applications on Windows NT. In *Proceedings of the 25th International Symposium on Computer Architecture (ISCA)*, pages 27–38, 1998.

[67] David A. Lifka. The anl/ibm sp scheduling system. In *IPPS '95: Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing*, pages 295–303, London, UK, 1995. Springer-Verlag.

[68] Witold Litwin, Marie-Anne Neimat, and Donovan Schneider. RP*: A Family of Order Preserving Scalable Distributed Data Structures. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB 20)*, pages 342–353, Santiago, Chile, September 1994.

[69] Michael J. Litzkow, Miron Livny, and Matt W. Mutka. Condor – A Hunter of Idle Workstations. In *Proceedings of ACM Computer Network Performance Symposium*, pages 104–111, June 1988.

[70] David T. Liu and Michael J. Franklin. GridDB: A Data-Centric Overlay for Scientific Grids. In *Proceedings of the 30th International Conference on Very Large Databases (VLDB 30)*, Toronto, Canada, September 2004.

[71] Steve Lohr. Unused PC Power to Run Grid for Unraveling Disease. *The New York Times*, November 16, 2004.

[72] Steve Lohr. New Group Will Promote Grid Computing for Business. *The New York Times*, January 24, 2005.

[73] Volker Markl, Guy M. Lohman, and Vijayshankar Raman. LEO: An autonomic query optimizer for DB2. *IBM Systems Journal*, 42:98–106, 2003.

[74] Ethan L. Miller and Randy H. Katz. Input/output behavior of supercomputing applications. In *Proceedings of the ACM/IEEE conference on Supercomputing*, pages 567–576, 1991.

[75] Ahuva W. Mu'alem and Dror G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Transactions on Parallel Distributed Systems*, 12(6):529–543, 2001.

[76] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A Read/Write Peer-to-Peer File System. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[77] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[78] John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the 10th ACM Symposium on Operating System Principles (SOSP '85)*, pages 15–24, Orcas Island, Washington, December 1985.

[79] Sang-Min Park, Jai-Hoon Kim, Young-Bae Ko, , and Won-Sik Yoon. Dynamic Data Grid Replication Strategy based on Internet Hierarchy. In *Second International Workshop on Grid and Cooperative Computing (GCC'2003)*, Shanghai, China, December 2003.

[80] Barbara K. Pasquale and George C. Polyzos. A static analysis of I/O characteristics of scientific applications in a production workload. In *Proceedings of the ACM/IEEE conference on Supercomputing*, pages 388–397, November 1993.

[81] Platform Computing. Improving Business Capacity with Distributed Computing. www.platform.com/industry/financial/, 2003.

[82] Florentina I. Popovici and John Wilkes. Profitable Services in an Uncertain World. In *HP Technical Report*, 2005.

[83] Arcot Rajasekar, Michael Wan, and Reagan Moore. MySRB and SRB - components of a data grid. In *In Proceedings of the Eleventh IEEE Symposium on High Performance Distributed Computing (HPDC)*, Edinburgh, Scotland, July 2002.

[84] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems (Third Edition)*. McGraw-Hill, 2004.

[85] Rajesh Raman. *Matchmaking Frameworks for Distributed Resource Management*. PhD thesis, University of Wisconsin-Madison, October 2000.

[86] Kavitha Ranganathan and Ian Foster. Decoupling computation and data scheduling in distributed data-intensive applications. In *Proceedings of the Eleventh IEEE Symposium on High Performance Distributed Computing (HPDC)*, Edinburgh, Scotland, July 2002.

[87] Luigi Rizzo. Dummynet: A Simple Approach to the Evaluation of Network Protocols. *ACM Computer Communication Review*, 27(1):31–41, 1997.

[88] Alexandru Romosan, Doron Rotem, Arie Shoshani, and Derek Wright. Co-Scheduling of Computation and Data on Computer Clusters. In *Scientific and Statistical Database Management Conference (SSDBM 2005)*, Santa Barbara, California, June 2005.

[89] Drew Roselli, Jacob R. Lorch, and Thomas E. Anderson. A Comparison of File System Workloads. In *Proceedings of the USENIX Annual Technical Conference (USENIX '00)*, pages 41–54, San Diego, California, June 2000.

[90] Emilia Rosti, Giuseppe Serazzi, Evgenia Smirni, and Mark S. Squillante. The impact of I/O on program behavior and parallel scheduling. In *Proceedings of the Joint International Conference on Measurement and Modelling of Computer Systems (SIGMETRICS)*, pages 56–65, 1998.

[91] Antony Rowstron and Peter Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.

[92] Marek Rusinkiewicz and Amit P. Sheth. Specification and Execution of Transactional Workflows. In *Modern Database Systems: The Object Model, Interoperability, and Beyond.*, pages 592–620. 1995.

[93] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[94] Asad Samar and Heinz Stockinger. Grid Data Management Pilot. In Proceedings of *IASTED International Conference on Applied Informatics (AI2001)*, February 2001.

[95] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the Migration of Virtual Computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, December 2002.

[96] Mahadev Satyanarayanan. A Study of File Sizes and Functional Lifetimes. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP '81)*, pages 96–108, Pacific Grove, California, December 1981.

[97] Tim Schoenharl, Scott Christley, and Douglas Thain. Patisserie: Support for Parameter Sweeps in a Fault-Tolerant, Massively Parallel, Peer-to-Peer Simulation Environment. In *Agent-Directed Simulation (ADS '05)*, San Diego, California, April 2005.

[98] Margo I. Seltzer, Yasuhiro Endo, Christoper Small, and Keith A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 213–228, Seattle, Washington, October 1996.

[99] Abraham Silberschatz and Peter Galvin. *Operating Systems Concepts*. Addison-Wesley, 1998.

[100] Quinn Snell, Mark Clement, David Jackson, , and Chad Gregory. The Performance Impact of Advance Reservation Meta-Scheduling. *Job Scheduling Strategies for Parallel Processing*, 1911, June 2000.

[101] Steven Soderbergh. Mac, Lies, and Videotape. www.apple.com/hotnews/articles/2002/04/fullfrontal/, 2002.

[102] W. T. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, and D. Anderson. A New Major SETI Project based on Project Serendip Data and 100,000 Personal Computers. In *Proceedings of the 5th International Conference on Bioastronomy*, 1997.

[103] A. K. Sum and J. J. de Pablo. Nautilus: Molecular Simulations code. Technical report, University of Wisconsin - Madison, Dept. of Chemical Engineering, 2002.

[104] Sun. Sun ONE Grid Engine Software. http://wwws.sun.com/software/gridware/, 2003.

[105] Edward Tenner. If Technology's Beyond Us, We Can Pretend It's Not There. *The Washington Post*, August 24, 2003.

[106] Douglas Thain and Miron Livny. Multiple bypass: Interposition agents for distributed computing. *Journal of Cluster Computing*, 4:39–47, 2001.

[107] Douglas Thain and Miron Livny. Parrot: Transparent User-Level Middleware for Data-Intensive Computing. In *Workshop on Adaptive Grid Middleware*, New Orleans, Louisiana, Sep 2003.

[108] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed Computing in Practice: The Condor Experience. *Concurrency and Computation: Practice and Experience*, 17(2):323–356, February 2005.

[109] The PBS Implementation Team. The Portable Batch System. http://www.openpbs.org/, 2002.

[110] Amin Vahdat and Thomas E. Anderson. Transparent Result Caching. In *Proceedings of the USENIX Annual Technical Conference (USENIX '98)*, New Orleans, Louisiana, June 1998.

[111] Amin Vahdat and Tom Anderson. Transparent result caching. Technical Report CSD-97-974, Computer Science Division, University of California-Berkeley, 1997.

[112] Sudharshan Vazhkudai, Steven Tuecke, and Ian Foster. Replica selection in the globus data grid. *IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, May 2001.

[113] Sudharshan S. Vazhkudai, Xiaosong Ma, Vincent W. Freeh, Jonathan W. Strickland, Nandan Tammineedi, and Stephen L. Scott. FreeLoader: Scavenging Desktop Storage Resources for Scientific Data. Technical report, Computer Science and Mathematics, Oak Ridge National Laboratory, 2005.

[114] Werner Vogels. File system usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 93–109, Kiawah Island Resort, South Carolina, December 1999.

[115] Richard Wolski. Dynamically Forecasting Network Performance using the Network Weather Service. *Cluster Computing*, 1(1):119–132, 1998.

[116] Frederick Wong, Richard Martin, Remzi H. Arpaci-Dusseau, David Wu, and David E. Culler. Architectural Requirements and Scalability of the NAS Parallel Benchmarks. In *Supercomputing '99*, Portland, Oregon, November 1999.

[117] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Shingh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA '95)*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

[118] Erez Zadok, Jeffrey Osborn, Ariye Shater, Charles Wright, Kiran-Kumar Muniswamy-Reddy, , and Jason Nieh. Reducing Storage Management Costs via Informed User-Based Policies. In *Proceedings of the 12th NASA Goddard/21st IEEE Conference on Mass Storage Systems and Technologies*, Adelphi, Maryland, April 2004.

[119] Songnian Zhou. LSF: Load Sharing in Large-scale Heterogeneous Distributed Systems. In *Proceedings of the Workshop on Cluster Computing*, Tallahassee, FL, December 1992.

APPENDIX

A.1 Validating the Simulator

We found external visualization to be extremely useful in its ability to validate the internal correctness of the simulator whose results we present in Chapter 4. To convince ourselves of this correctness we required evidence of both functional correctness of the simulator as well as its ability to produce (*and reproduce)* reasonable performance estimates. Specifically, we wanted validating evidence about four features of the simulator: one, the functional correctness of its representation of the workload, two, the functional correctness of the compute system, three, the functional correctness of the scheduler, and finally evidence of the accuracy of its simulated results.

A.1.1 Validating Functional Correctness of the Workload

To validate the functional correctness of the simulator's ability to interpret a workload, we create a synthetic workload with a known set of data and job dependencies and then use the *DOT* graph layout software to verify that the scheduler has interpreted the workload correctly.

Shown in Figure A.1 is a recreation of a workload which we check to verify that the simulated scheduler has properly interpreted inter-job dependencies as well as the mount points between volumes and jobs. The circles are jobs and the double-edged quadrangles are volumes; the inverted trapezoids are volumes accessed by only a single pipeline (*i.e.* endpoint and pipeline) and the double-edged rectangles are batch volumes shared across pipelines. The arrows show the dependencies between jobs and the mount points linking jobs and volumes. The percentage numbers in the volumes are the amount of the total storage capacity of the system needed for that data. In this example are shown two pipelines consisting of three jobs each. Each pipeline has an input endpoint volume (*e.g.* iA1) mounted by the first job and pipeline volumes shared between jobs in the pipeline (*e.g.* pA2). The first job in each pipeline also reads from batch volume B1. Also shown as single-edged rectangles are the extracted data which is moved from each pipeline to the home node following successful completion (*e.g.* bar.A).

Although it is not shown here, it is additionally possible to view a sequence of these workload visualizations which use color to indicate the state of the jobs and volumes (*e.g.* a job can be colored green for ready, blue for running and black when finished). Examining such a sequence of these workload "snapshots" shows the changing state of the workload as its individual jobs are executed.

A.1.2 Validating Functional Correctness of the Compute System

In addition to verifying the more coarse-grained correctness of the simulator's interpretation of the workload, we used visualization to confirm the correctness of the much more finely-grained data flow across the diverse components of the modeled system. This visualization was slightly more complex to develop as we were unable to leverage pre-existing tools. Using Perl and the tk imaging libraries, we wrote a viewer to display each of the machines in the system, and the state of each of their modeled components (*i.e.* network, memory, and disk). A screen capture of this tool is shown in the left side of Figure A.2. Each rectangle represents a different machine in the environment; one for the home storage server and one for each compute node. Within each machine representation are drawn the queues for each of the components on that machine as well as summary information for the memory and disk contents.

```
job A.1 datafiles/jobA.1
job A.2 datafiles/jobA.2
job A.3 datafiles/jobA.3
job B.1 datafiles/jobB.1
job B.2 datafiles/jobB.2
job B.3 datafiles/jobB.3

parent A.1 child A.2
parent A.2 child A.3
parent B.1 child B.2
parent B.2 child B.3

volume i.A.1 /home/in/   10MB
volume i.B.1 /home/in/   10MB
volume B.1   /home/data/ 10MB
volume p.A.1 scratch      5MB
volume p.A.2 scratch      5MB
volume p.A.3 scratch      5MB
volume p.B.1 scratch      5MB
volume p.B.2 scratch      5MB
volume p.B.3 scratch      5MB

mount i.A.1 A.1 /input/
mount i.B.1 B.1 /input/
mount p.A.1 A.1 /out/
mount B.1   A.1 /data/
mount p.A.1 A.2 /in/
mount p.A.2 A.2 /out/
mount p.A.2 A.3 /in/
mount p.A.3 A.3 /out/
mount B.1   B.1 /data/
mount p.B.1 B.1 /out/
mount p.B.1 B.2 /in/
mount p.B.2 B.2 /out/
mount p.B.2 B.3 /in/
mount p.B.3 B.3 /out/

extract p.A.3 bar.A.3 bar.A
extract p.B.3 bar.B.3 bar.B
```

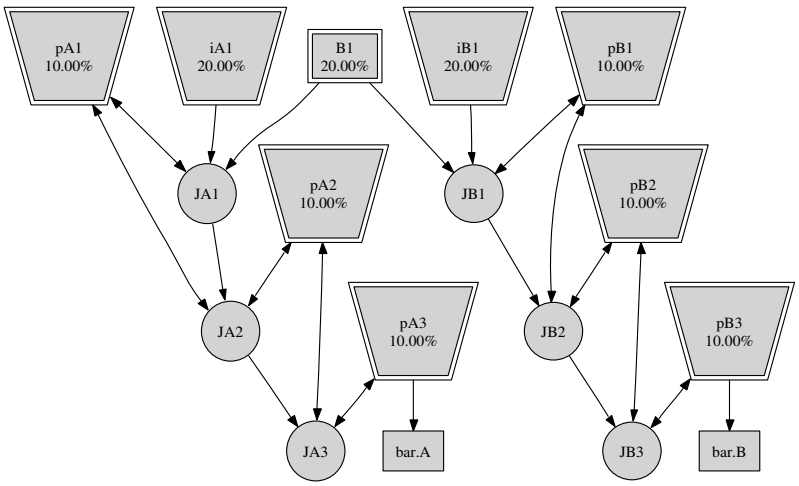


Figure A.1: **Validating Functional Correctness of the Workload Representation.** *This illustration shows that the simulator creates an accurate internal representation of the workload as shown on the right from the description shown on the left.*

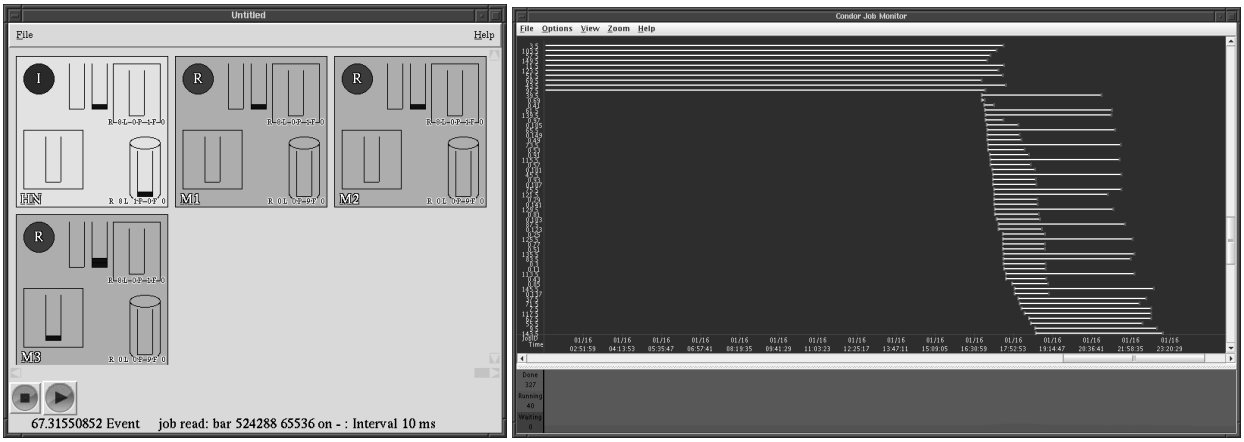
This system viewer reads an event trace log of the simulator and displays the state of the system as each event is executed. This tool proved particularly useful as a debugging tool that could be used to trace the activity within the system that resulted from various events. For example, to verify that data was moved correctly to satisfy a particular job read event, the viewer could be used to ensure that this job read would result in a network request which moved from network buffer to network buffer and was followed by a disk read, a network transfer, and a memory write before the job could progress further.

A.1.3 Validating Functional Correctness of the Scheduler

On the right side of Figure A.2 is shown the validation of the scheduler's ability to correctly observe inter-job dependencies. To acquire this visualization, we wrote a small script to convert our log files to match the formatting of user log files created by the Condor batch scheduling system. We were then able to leverage pre-existing software for visualizing these user job logs. As shown in the screen capture, this visualization tool represents each job in the workload as a horizontal line. For this specific validating experiment, we ran a workload with 30 pipelines of depth two. As seen here, the scheduler correctly delays the execution of half of the jobs until their dependent parent has finished executing.

A.1.4 Validating Performance Accuracy

Having validated the functional validity of the simulator, we examine its ability to produce realistic and meaningful performance estimates. First we discuss two different experiments that show reasonable and intuitive performance profiles for different I/O techniques and across multiple cache hierarchies. We then conclude our validation



Compute system Scheduling plan

Figure A.2: **Validating Functional Correctness of the Compute System and the Scheduler.** *The illustration on the left shows a representation of the compute system including the disks, buffer caches, and network queues for each machine. On the right is a visualization of the execution schedule for a workload with inter-job dependencies.*

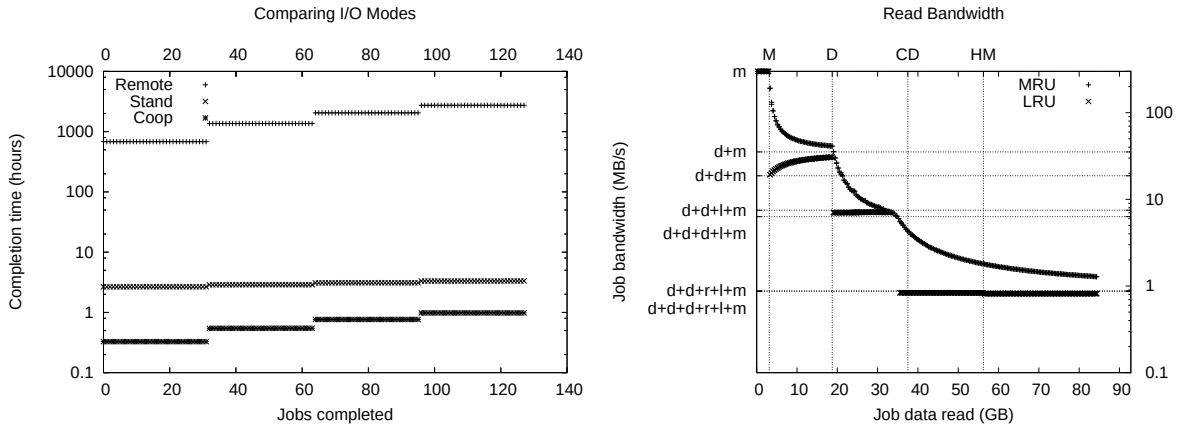


Figure A.3: **Validating the Performance Correctness of the Simulator.** *These two graphs show validating experiments for the performance of the simulator. On the left are the completion times for three different I/O techniques of binding jobs to the home storage server. On the right is a comparison of MRU and LRU replacement policies across a hierarchy of caches in the compute cluster.*

with a direct comparison of the simulator against previously acquired results using the full BAD-FS implementation.

Figure A.3 shows two experiments run to check this. The first graph on the left shows synthetic BLAST jobs being executed on a cluster of 32 compute nodes using three different methods of linking the jobs to their batch data. Using remote I/O, all jobs went directly (and simultaneously) to the home node to read the batch data. In stand-alone mode, the first job run on a compute node fetches batch data from the home node and then caches it; subsequent jobs then read from the cache. Note that stand-alone mode approximates the behavior of AFS. Finally, in cooperative caching mode as is done in BAD-FS, the compute nodes cooperatively fetch the batch data from the home nodes for the first 32 jobs (*i.e.* the number of compute nodes) and then subsequent jobs read from the cached data. Because of the large disparity in performance between remote I/O and the other two I/O modes, the y-axis is shown in log-scale.

As expected, the performance using remote I/O is by far the worst and there is no difference in run-times

for any of the different sets of 32 jobs. Comparing stand-alone caching and cooperative caching yields slightly different conclusions. The first set of jobs that run when the cache is cold show a difference in run-time as the stand-alone mode results in contention for the network bandwidth of the home node. Conversely, in cooperative caching, the compute nodes effectively share this bandwidth by multi-casting the batch data from the home node. Once the data is cached however, the performance is the same.

The right-most graph in Figure A.3 provides additional validation by showing the bandwidth achieved by batch-intensive jobs run on a "warm" system consisting of two compute nodes. The jobs scan their batch data in a linear manner and the two different lines show the difference between using MRU and LRU replacement policies. The y-axis shows the bandwidth for each job and the x-axis is the amount of batch-data read by each job. Along the top of the x-axis are labels for the various cache capacities of system components. `M` marks the size of the buffer cache for each of two compute nodes in the system; `D` is the size of the disk on each of the compute nodes; `CD` is the size of the cooperative disk composed of the two disks of the compute nodes and finally `HM` is the size of the buffer cache on the home node. The lower-case tics on the left y-axis represent the various bandwidths of the different components. Tic marks which show the sums of these various bandwidths represent the serial bandwidths for data which must pass serially through all of those components. For example, when the batch data read by jobs is less than `M`, then the warm jobs realize a bandwidth of `m` because all of their data fits within the buffer cache of each compute node and warm jobs can be satisfied entirely at memory bandwidth.

As the amount of batch-data read by the jobs increases beyond `M`, the buffer caches can no longer hold all batch data and more of this data must be read from the disk. In this case however, the bandwidth achieved by a job is not merely the bandwidth of disk but rather the serial bandwidth of moving data off disk, into memory and then copying it into the address space of the job (*i.e.* thus `d+m` and not `d`). As expected for jobs which linearly scan their data, MRU performance more gracefully degrades whereas LRU performance drops more drastically. Similar effects are noticed as the data read by each job surpasses the size of each disk cache and then the size of the cooperative disk cache and finally the size of the buffer cache on the home node. Note that `l` is the local network bandwidth and that `r` is the remote bandwidth between the compute nodes and the home node.

One interesting effect is seen for the LRU replacement policy for jobs which read an amount of batch data between `M`, the amount of buffer cache on the compute nodes, and `D`, the size of disk cache of the compute nodes. In this region, performance actually *improves* as the amount of data approaches `D`. This effect is due to the nature of the buffer cache replacement policy. As blocks are retrieved from disk, they must necessarily replace blocks currently in the cache. If the victim block is dirty, it must first be written to disk before the new block can be read into memory. In this scenario, the bandwidth is expected to be `d+d+m`, two disk I/O's followed by one memory read, instead of the more intuitive `d+m`. When the amount of batch data only slightly exceeds `M`, then as the warm jobs first run, the majority of blocks in the buffer cache are dirty. Thus for each missing block (which is all of them when LRU is used), a dirty block must first be flushed before the new block can be read. Conversely, as the amount of batch data approaches `D`, more of the dirty blocks will have already been flushed during the execution of the cold jobs. Thus as new blocks are requested, a smaller percentage of the victim blocks will need to be flushed and achieved bandwidth approaches `d+m`.

What is important to take away from these results is not any one observation about the relative performance of LRU and MRU cache replacement policies but rather the verisimilitude of the simulation framework. These results in particular show that the caching hierarchy of the compute cluster performs as expected in regards to when and how data is evicted and the measured bandwidths match the configuration as well.

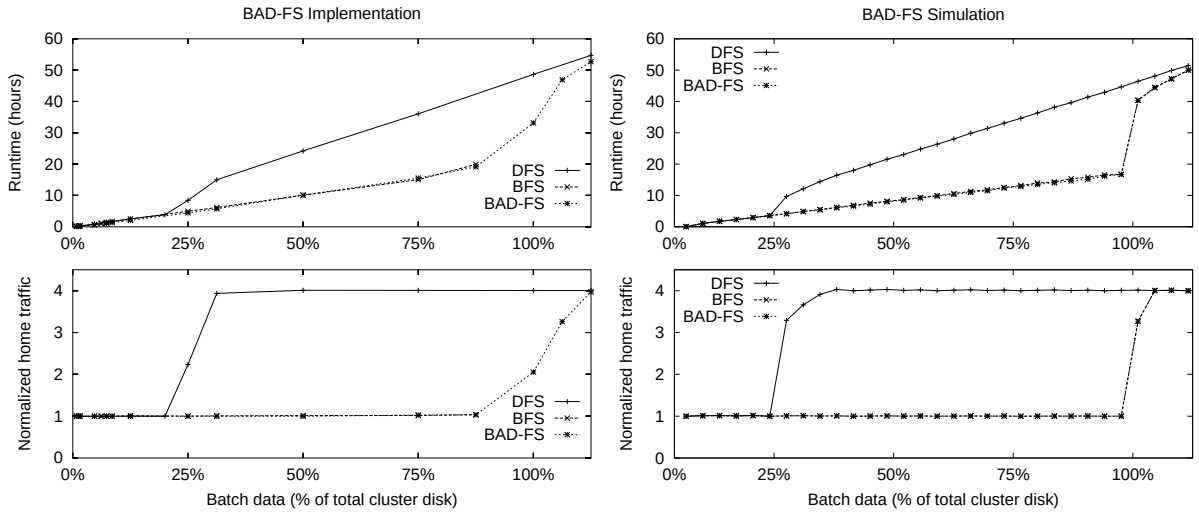


Figure A.4: **Checking Simulator Performance against the BAD-FS Implementation.** *The graphs on the left are the same as were discussed previously in Figure 3.4 of Chapter 3 and show the runtime and normalized traffic to the home node incurred when running a synthetic batch-intensive workload using three different workload traversals. On the right are the results obtained for a recreation of this experiment within our simulation framework.*

A.1.5 Verifying Against the BAD-FS Implementation

Finally, in Figure A.4, we simulate an experiment run previously using the BAD-FS implementation as shown and explained in more detail in Figure 3.4 of Chapter 3. On the left side, are the runtimes and the normalized amounts of wide-area traffic between the home storage server and the compute cluster as seen when using the BAD-FS implementation across a compute cluster of sixteen nodes at the University of Wisconsin. On the x-axis, we vary the size of batch data accessed by each job in a synthetic batch-intensive workload of 32 pipelines of depth four.

Compared within each graph are the results obtained using either a strict breadth-first traversal of the workload, a depth-first traversal or a capacity-aware traversal as is done by the BAD-FS scheduler. When the total amount of batch data fits within the storage available on the cluster (*i.e.* when each of the four batch volumes is less than 25%), each of the three traversals achieves the same results. However, once the size of the batch volumes exceeds 25%, then the depth-first traversal will overallocate the available storage thereby causing the batch data to be evicted and then refetched for the execution of subsequent pipelines. This is seen in both the runtime and normalized traffic graphs as the depth-first schedule deviates from the other two at approximately 25%. The maximum value for the normalized amount of traffic is four because with 64 pipelines and 16 compute nodes, each server executes four pipelines. Thus without careful capacity planning, the worst case utilization of the wide-area network will result in the batch data being redundantly fetched for each executing pipeline.

For this synthetic batch-intensive workload, the capacity-aware planning of the BAD-FS scheduler results in a breadth-first traversal and thus the values for both breadth-first and BAD-FS remain the same and outperform the depth-first traversal while the size of the batch data is between 25% and 100% of the total cluster storage. Notice however that once the batch data exceeds the available storage that caching the data is no longer possible and all three traversals are forced to redundantly refetch the batch data and suffer the corresponding reduction in runtimes.

On the right side, we compare the results achieved by recreating this experiment within our simulation framework. We recreate the synthetic workloads and configure a compute environment using the same capacities used

in the implementation experiment and the same bandwidths as were measured then. Although the results for both runtime and normalized traffic appear extremely similar, there are two slight differences. The first is the slope of the "knees" of the curves are smoother in the simulated graphs. This is due to their being more measured values in the simulated graphs which attests to the relative ease of using the simulator as opposed to actually running these experiments on actual machines using the BAD-FS implementation. This relative ease is due to both needing fewer machines as well as running more quickly. Each simulated point of data required less than an hour of computation on a single machine whereas each point of data collected using the full implementation required up to fifty hours of computation across sixteen compute nodes, a seventeenth machine for the home storage node and an eighteenth for the scheduler.

The second difference between the simulator and the implementation is hard to see in the runtime graphs but is obvious when looking at the actual data. This difference is that the runtimes reported by the simulator are slightly faster than those achieved by the implementation. This difference is likely due to not modelling the scheduling latency within the simulator. When a job finishes executing in the actual implementation, the notification of its completion must travel across the network from the compute node to the scheduler which then must decide which job to schedule next and then send that job back to the now idle compute node. However, for such long running jobs like those modelled here, this scheduling latency should be almost entirely amoritized by the runtimes of the jobs. As expected this is the case, resulting in the very close approximation of the simulated performance to the real.