Challenges in Building a Two-Tiered Learning Architecture for Disk Layout

Brandon Salmon, Eno Thereska, Craig A.N. Soules, John D. Strunk,

Gregory R. Ganger

August 2004

CMU-PDL-04-109

School of Electrical and Computer Engineering

Carnegie Mellon University

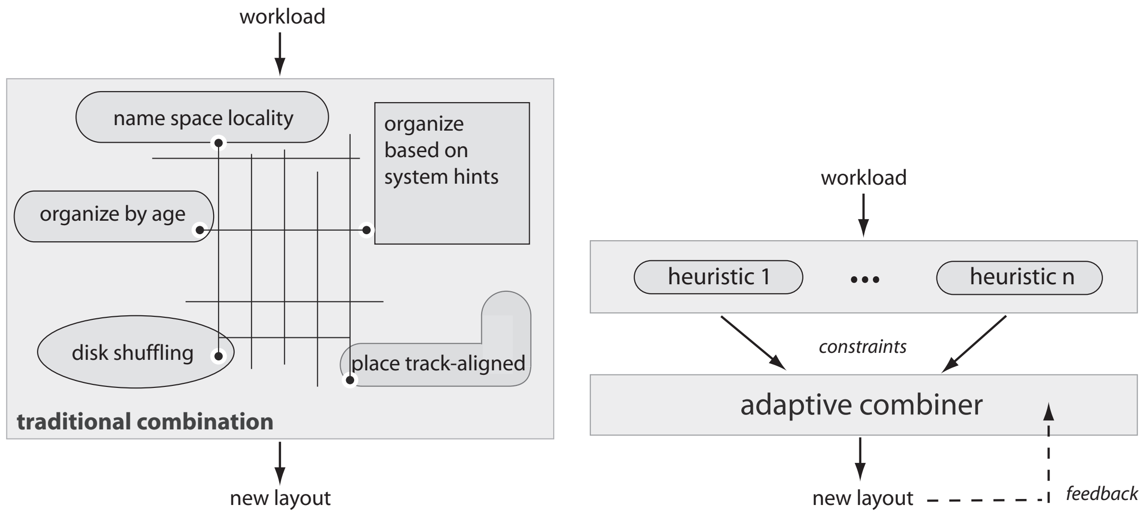
Pittsburgh, PA 15213

Abstract

Choosing the correct settings for large systems can be a daunting task. The performance of the system is often heavily dependent upon these settings, and the "correct" settings are often closely coupled with the workload. System designers usually resort to using a set of heuristic approaches that are known to work well in some cases. However, hand-combining these heuristics is painstaking and fragile. We propose a two-tiered architecture that makes this combination transparent and robust, and describe an application of the architecture to the problem of disk layout optimization. This two-tiered architecture consists of a set of independent heuristics, and an adaptive method of combining them. However, building such a system has proved to be more difficult than expected. Each heuristic depends heavily on decisions from other heuristics, making it difficult to break the problem into smaller pieces. This paper outlines our approaches and how they have worked, discusses the biggest challenges in building the system, and mentions additional possible solutions. Whether this problem is solvable is still open to debate, but the experiences reported provide a cautionary tale; system policy automation is complex and difficult.

We thank the members and companies of the PDL Consortium (including EMC, Engenio, Hewlett-Packard, HGST, Hitachi, IBM, Intel, Microsoft, Network Appliance, Oracle, Panasas, Seagate, Sun, and Veritas) for their interest, insights, feedback, and support. Their work is partially funded by the National Science Foundation, via grant #CCR-0326453.

Keywords: disk layout, adaptive, self-managing, self-tuning, learning, automated tuning



(a) Traditional Monolithic Architecture.

(b) Two-tiered Architecture.

Figure 1: **Two-tiered vs. traditional architecture for adaptive layout software.** The traditional architecture combines different heuristics in an ad-hoc fashion, usually using a complicated mesh of if-then-else logic. This combination is difficult to build and tends to be fragile. The two-tiered architecture separates the heuristics from the combiner and uses feedback to refine its decisions and utilize the best parts of each heuristic. This provides a flexible and robust way to combine a set of heuristics.

1 Introduction

Internal system policies, such as on-disk layout and disk prefetching, have been the subject of decades of research. Researchers try to identify algorithms that work well for different workload mixes, developers try to decide which to use and how to configure them, and administrators must decide on values for tunable parameters (e.g., run lengths and prefetch horizons). Unfortunately, this process places significant burden on developers and administrators, yet still may not perform well in the face of new and changing workloads.

To address this, researchers now strive for automated algorithms that learn the right settings for a given system. Of course, different configurations work best for different workloads meaning that any particular setup will work well for one workload and poorly for others. Worse, most deployed systems support many workloads simultaneously, potentially making any single decision suboptimal for the aggregate. Devising a composite algorithm for such circumstances can be a daunting task, and updating such an algorithm to workload changes even more so.

This paper describes a two-tiered architecture for such automated self-tuning software, using on-disk data layout as a concrete example [24]. Instead of a single monolithic algorithm, as illustrated in Figure 1a, the decision-making software consists of a set of independent *heuristics* and an *adaptive combiner*, used for merging the heuristics' suggested solutions as shown in Figure 1b. Each heuristic implements a single policy that hopefully works well in some circumstances but not necessarily in all. Heuristics provide suggested *constraints* on the end layout, such as placing a given block in a given region or allocating a set of blocks sequentially. The adaptive combiner uses prediction models and on-line observations to balance and merge conflicting constraints.

This two-tiered architecture would provide at least three benefits. First, heuristic implementations could focus on particular workload characteristics, making local decisions without concern for global consequences. Second, new heuristics could be added easily, without changing the rest of the software. In fact, bad (or poorly implemented) heuristics could even be handled, because their constraints would be identified as less desirable and ignored. Third, the adaptive combiner could balance constraints without knowledge of or concern for how they were generated. The overall result would be a simpler and more robust software structure.

However, in order for this architecture to work, an efficient learning agent must be developed to actually do the combination of the heuristics. While the problem seems reasonably solvable at first glance, it is actually quite difficult. This paper describes our experiences in trying to build such a system and lessons learned. It presents several failed attempts to solve it, and analyzes why they did not work. It remains an open question whether the problem is solvable, but our efforts highlight an important point: policy automation is difficult in practice.

The remainder of this paper is organized as follows. Section 2 discusses related work. Sections 3 and 4 discusses the overall design of the system. Section 5 outlines the experimental set up for the paper, and Section 6 describes an evaluation of the system. Section 7 describes some of the challenges in building a functional system, and Section 8 describes various tradeoffs in building the system. Sections 9 and 10 describe future work and conclusions.

2 Related Work

The AI community continues to develop and extend the capabilities of automated learning systems. The systems community is adopting these automated approaches to address hard problems in systems management. This section discusses related work, both from the AI and systems perspectives.

AI techniques: The AI community has long recognized the need for self-managing systems. In fact, a whole branch of AI research, machine learning, exists especially to solve real-life problems where human involvement is not practical [17].

One general AI problem of relevance is the *n-experts problem*, in which a system must choose between the outputs of n different experts. The n-experts problem is not an exact match to our problem, because we am merging experts' suggestions rather than choosing one. Nonetheless, solutions such as the weighted majority algorithm [12] provide valuable insight.

Another general challenge for the AI community is the efficient exploration of a large state space (i.e., the set of all possible solutions to an optimization problem). For example, our current prototype explores its state space using a variety of guided hill-climbing algorithms and a method similar to simulated annealing to avoid local maxima.

Genetic Algorithms [6] are another approach to optimizing problems without having a detailed understanding of the underlying trade offs. The method works by choosing a population of solutions, evaluating their effectiveness, and choosing a new population from those solution which perform well, and mutations and combinations of these solutions. While our problem could be adapted to a genetic algorithm, we fear it might take more than the 50-100 iterations that are feasible in our current system, given the approach's lack of knowledge of the problem specifics. However, exploration of this technique would be interesting.

Adaptive disk layout techniques: A disk layout is the mapping between a system's logical view of storage and physical disk locations. Useful heuristics have been devised based on block-level access patterns and file-level information.

Block-based heuristics arrange the layout of commonly accessed blocks to minimize access latency. For example, Ruemmler and Wilkes [23] explored putting frequently used data in the middle of the disk to minimize seek time. Wang and Hu [29] tuned a log-structured file system [21] by putting active segments on the high bandwidth areas of the disk. Several researchers [1, 13, 18, 30] have explored replication of disk blocks to minimize seek and rotational latencies. Hsu et al. propose the ALIS system [9], which combines two disk layout heuristics, heat clustering and run packing. The project shows a good hand-crafted combination of two heuristics, but does not attempt to combine an arbitrary set of conflicting heuristics.

File-based heuristics use information about inter and intra-file relationships to co-locate related blocks. For example, most file systems try to allocate blocks of a file sequentially. C-FFS [5] allocates the data blocks that belong to multiple small files named by the same directory adjacently. Hummingbird [10] and Cheetah [27] perform similar grouping for related web objects (e.g., an HTML document and its embedded images).

Other system management policy techniques: Relevant research has also been done on storage system policies, such as caching and prefetching [3, 19]. Similar schemes have been used in other domains as well, such as branch prediction in modern processors [16, 28]. For example, in NOAH [2], Amer et al. extend the first and last successor models for predictive caching. It provides a way to determine which of these models is relevant, gaining the benefits of both. In [11], the authors explore the use of a trie to store sequences of file system events, and their frequencies, thereby establishing multiple contexts to make prefetching decisions. Griffioen and Appleton in [7] and [8] record file events in a probability graph. Each node in the graph is a file; directed edges from each node to other nodes represent related accesses, and edges are weighted by frequency of occurrence. Prefetching decisions are then made by choosing accesses with the largest probability, conditioned on the fact that the file being prefetched for has just been opened.

Madhyastha and Reed [15, 14] propose a system that optimizes input/output performance by recognizing file access patterns and choosing appropriate caching policies to match application needs. The authors also discuss the concept of *adaptive steering* which uses a feedback system to determine what file system policy to use. They claim that *a priori* identification of file system policy is impossible because policies may depend on data sets or unforeseeable environments, and inaccurate hints may worsen the problem. To classify each access pattern into a category, each of which corresponds to a different file policies, the authors used a pre-trained neural network. Their approach is similar to ours in that it uses workload information to decide between a set of algorithms, but it differs in several important ways. Madhyastha and Reed are changing caching behavior instead of actually redoing disk layout as explored here. Their system works on a per-process basis, while our system analyzes global behavior. In addition, they use a pre-trained neural network, while we propose online learning. Finally, they automatically assign a given type of workload to an optimization algorithm, while we attempt to learn this association.

To summarize, researchers have explored a variety of heuristics for disk layout optimization, as well as hand-crafted combinations of heuristics. In other policy decisions they have explored methods to choose from a set of heuristics. However, this work is the first to explore a general combination method for heuristics in disk layout.

3 Two-tiered Learning For Layout

This work focuses on the problem of identifying a disk layout that improves performance for a given workload. At the most general level, this is an optimization problem that takes as input a workload and outputs a new layout. However, due to the size of the state space, solving this problem using a brute-force optimization algorithm is intractable; a typical disk has millions of blocks, and workloads often contain millions of requests.

For this reason most approaches take a much simpler view of the problem, and only implement hand-crafted heuristics. These heuristics will improve performance if the assumptions they make are true, but will

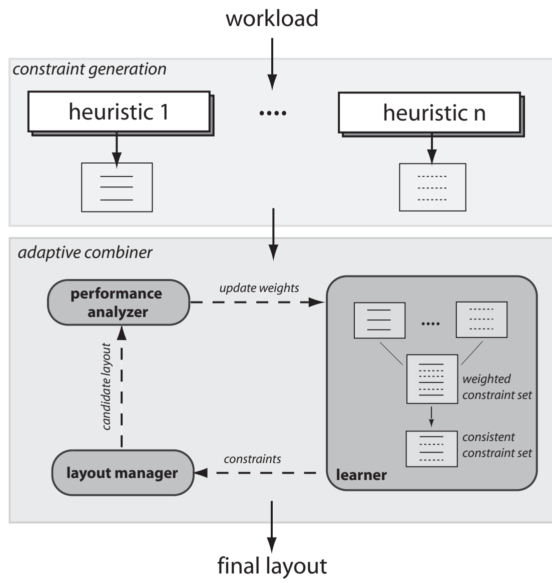


Figure 2: **The two-tiered learning architecture.** This figure illustrates the two components of a two-tiered architecture for refining disk layout. The *constraint generation* layer consists of the individual heuristics and their constraints. The *adaptive combiner* layer merges the constraints and refines the resulting data layout.

not work in different cases and may miss opportunities for other optimization.

The two-tiered learning architecture takes a middle ground, by using heuristics to build up a smaller, more directed state space. The system then searches for better performing disk layouts within this space. This makes the system more flexible than the single heuristic approach, yet keeps the problem tractable, hopefully leading to a manageable state space.

In effect, the two-tiered architecture allows the system designer to prune the search space with a set of well-known heuristics. This makes the problem more tractable, while still allowing an adaptive solution. Finding the balance between flexibility and problem size is a key point of this kind of system. If the field is too narrow, (e.g. too few heuristics) the system will be unable to find the good solutions because they will already have been eliminated. However, if the state space is allowed to grow too large (e.g. too many heuristics), the system will be unable to find a good solution because it is overwhelmed.

4 Design

Figure 2 illustrates the two-tiered learning architecture. The constraint generation layer consists of a collection of independent heuristics. Each of these heuristics generates a set of *constraints*, or invariants, based on the workload. The adaptive combiner consists of three parts: the learner, the layout manager, and the performance analyzer.

Constraint Type	Description
place-in-region	place blocks into a specific region
place-seq	place blocks sequentially

Table 1: **Common Constraint Language.** This table shows the constraints we use in our implementation.

The *learner* assigns a value measurement, called a weight, to each of the constraints based on the performance of previous disk layouts. It then uses the weights to decide which constraints to apply, and then to resolve any conflicts between them, creating a single set of consistent constraints.

The *layout manager* takes the constraint set from the learner and builds a new disk layout.

The *performance analyzer* takes each candidate layout and determines its success based on the target performance metrics and the given workload. We use the performance metric of average response time in this paper. The results of the performance analyzer are then passed to the learner for use in updating the constraint weights.

The adaptive combiner iterates through these steps to refine the layout. In order to search the state space for the maximum allowed time, the adaptive combiner holds the best observed layout. If at any time the adaptive combiner must be stopped (e.g., due to computation constraints or diminishing returns), it can immediately output the best observed layout.

The remainder of this section describes the heuristics, learner, layout manager, and performance analyzer in more detail.

4.1 Heuristics

The first level of the architecture is the heuristic layer. At this layer, a group of independent heuristics propose changes to the disk layout to improve performance. A prerequisite to doing so is having a common language in which to express heuristic suggestions.

4.1.1 Common Constraint Language

The learner uses *constraints* as the common language used by disk organization heuristics. A constraint is an invariant on the disk layout that a heuristic considers important. A constraint may be specified on a single disk block or a set of blocks. The learner combines heuristics by selectively applying their constraints.

Table 1 shows two example constraints. *Place-in-region* constraints specify in which region of the disk a set of blocks should be placed.[1] *Place-seq* constraints specify a set of blocks that should be placed sequentially. These two constraint types have been sufficient for many heuristics, but additions are expected for some future heuristics. For example, heuristics that exploit replication and low-level device characteristics (e.g. track-aligned extents [26]) may require additional constraints.

By dividing each heuristic's suggestions into constraints, the learner can apply the pieces of the heuristic that are helpful and ignore those that are not. This allows for a finer granularity in dealing with the problem, but also increases the size of the problem. Experimentation into ways to decrease the granularity of the problem, such as dealing with reorganization at a file level could be very interesting. There is a trade-off between finding a granularity that effectively captures the patterns of the workload, without exposing the learner to too many options.

4.1.2 Example Heuristics

This section describes the five heuristics currently used in the prototype constraint generation layer.

Disk Shuffling: The disk shuffling heuristic generates *place-in-region* constraints to place frequently accessed data near the middle of the disk and less frequently accessed data towards the edges [23]. Such an arrangement reduces the average seek distance of requests. This heuristic is done at a large granularity, to allow for sequential placement of data within the disk shuffling arrangement.

Threading: The threading heuristic generates *place-seq* constraints for sets of blocks usually accessed in a sequential manner. Doing so exploits the efficiency of disk streaming. It is implemented using a first-order Markov-model [20] that tracks the probability of one block following another.

Run clustering: The run clustering heuristic is similar to the threading heuristic, but it is built to find sequences in a multi-threaded environment. In this environment, blocks in a sequence may not occur immediately after one another, but may have other blocks interspersed in the sequence. The run packing heuristic keeps track of a window of previous requests, weighting requests closer in the sequence more highly than those farther away [9].

Front loading: This heuristic generates *place-in-region* constraints to place frequently accessed data at the beginning of the disk, where the streaming bandwidth of the disk is higher (due to zoned recording.)

Bad: This heuristic is used purely for testing the adaptive combiner. It is built to hurt performance, to simulate a poorly implemented or poorly fitted heuristic. It generates *place-in-region* constraints to spread the most frequently accessed blocks across the disk in an attempt to destroy locality. We do not plot results for the bad heuristic, because it increases response times by at least an order of magnitude.

[1] The system currently divides the disk into 11 regions to simplify placement constraints.

4.2 Learner

The learner is the most complicated portion of the system. It is the learner's job to decide which constraints to apply and which to ignore. This section discusses the basic problem the learner seeks to solve, and an overview of the methods we have used to attempt to solve it.

4.2.1 Problem Overview

The learner considers the problem of optimizing the response time for a given set of trace data. We use a time period of one day's worth of trace, although the time period could be modified. We are seeking to optimize the sum of the response times for the trace segment, $R = \sum_{d \in D}(r_d)$, where r_d is the response time of request d, and D is all of the requests in the trace. This can be stated as a block-centric problem by formulating R in terms of blocks: $R = \sum_{b \in B}(R_b)$, where R_b is the sum of the response times of all accesses to block b, and B is the set of all blocks accessed by the trace.

However, because the learner is deciding whether to apply constraints, we convert the problem into a constraint-centric problem. To do this conversion, we assign weights to the constraints to approximate the block-centric weights. The weight of a constraint W_c is the sum of the response times of each of the blocks that the constraint places, or $W_c = \sum_{b \in B} R_b$, where B is the set of blocks contained in the constraint. In other words, the weight of a constraint is the sum of the response times for the requests to all of the blocks it affects. The learner then uses a variant of this constraint weight, which we will call the *constraint potential*, to determine whether to apply each constraint. This constraint potential is a measure of the actual usefulness of the constraint. Section 8.3 discusses several methods of computing constraint potential in detail.

An alternate way of formulating the problem, that we have not explored, is to keep the problem in a block-centric form. In this formulation, the response time of each set of blocks can be written as $R_b = f(c_1, c_2, ..., c_n)$ where c_i is a boolean variable indicating whether the ith constraint affecting this block is applied. This approach is an interesting alternative that warrants exploration.

4.2.2 Conflicts

One difficulty inherent in the problem is constraint conflicts. Because of their independence, different heuristics may generate conflicting constraints (e.g., the front loading heuristic places blocks on the outer tracks, while disk shuffling places them near the center of the disk). The learner must resolve these conflicts, choosing which constraints to apply and which to ignore. Figure 3 shows some example conflicts. The figure shows how conflicts can involve just two constraints, or many constraints.

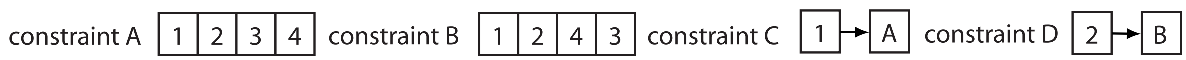


Figure 3: **Conflict example.** This figure illustrates conflict examples. In this example, the sequential constraints A and B conflict; only one of them can be applied. This is a conflict of length one. In addition, if either of the sequential constraints is applied, only one of the two placement constraints C and D can be applied. This second conflict is a conflict of length 2.

These conflicts are dependent on both the workload and the heuristics being used. Section 7.1 discusses conflicts in more detail, and quantifies their frequency.

4.2.3 Method

We approach the optimization problem using a variety of heuristic approaches. We expected that the interdependencies in the problem would be loose enough, and the gains high enough, to allow these ad-hoc approaches to have some success. The details of each individual learning approach are found in Section 8.3. Each approach has a slightly different way of computing the constraint potential, but all of the approaches share a similar method.

Each method uses the constraint potential to determine which constraints to apply. Once it has chosen a set of constraints to apply, it sorts them by constraint potential and applies them greedily. This assures that if two constraints conflict, the constraint with a higher potential will be applied.

Each approach will also try to apply seemingly sub-optimal constraints occasionally to test new options and avoid local minima. While this method is certainly not an optimal solution, it is a reasonable first try at the problem. Though the problem is too large to allow exhaustive search (the number of constraints is usually in the thousands), there are other more sophisticated approaches that would be interesting to apply.

4.2.4 Test window

The learner is looking for constraints that not only perform well on the given day, but also for at least several days in the future. Section 7.2 discusses this in detail. To accomplish this goal, the learner will not actually apply a constraint until after it is tested on several days. This is called the *test window*. The test window used in this paper is 3 days.

For each constraint the learner also keeps a cumulative weight which is the sum of the constraint potential for the last several periods. After *test window* periods, if the constraint has a positive cumulative weight, meaning it would have been a net gain over the last few periods, it will be applied. Otherwise, it will be discarded.

4.3 Layout Manager

Once the learner has decided which constraints it would like to apply, the layout manager must convert these constraints into an actual disk layout.

The current implementation assumes that the layout manager knows which blocks are in use and which are free. In the trace, this is accomplished by simulating a larger disk than that used in the trace. All of the experiments use a disk that is 25% unallocated. This free block knowledge allows for more intelligent reorganization, because the system can move blocks into free space, instead of having to disrupt the location of live blocks.

The layout manager uses several steps to convert the constraints into a layout. First, it sorts the constraints into the areas into which they must be placed. If constraints are not placed in any area, it will place them in the first open area starting at the beginning of the disk.

Once it has sorted the constraints into disk areas, it sorts the constraints in each disk area by the first logical block number it affects, to try to keep as similar an order as possible to the previous layout. Once it has done this, it attempts to place all of the newly placed blocks into empty spaces in the disk area. If that is not possible, it de-fragments the area and places the new blocks at the end of the area.

If there is not enough empty space in the area, the layout manager will move a chunk of contiguous blocks from the area to another location of the disk in order to free up enough space for the new blocks. It will try to move this data to an area as close to the original area as possible.

This method helps avoid the problem of incremental change. If the layout manager is not careful, it may undo work that has been done previously by the learner. For example, a naive approach would be to place blocks in the beginning of each area, but this means that the blocks placed in the last iteration would be disturbed each day.

A completely accurate solution would have to maintain the constraints which have been applied previously in order to avoid breaking them by the application of new constraints. This is infeasible because of the amount of storage necessary. The current method attempts to avoid breaking many constraints by only moving large chunks of the disk at a time and trying to move the data nearby.

5 Experimental Set Up

This section describes much of the common set up for the experiments found in this paper. All of the experiments are done in simulation using Disksim [4] with parameters for a Seagate Cheetah 36ES 18GB disk drive extracted using the DiXTrac tool [25]. All of the experiments in the paper use a block size of

4KB, since most file systems will do operations on scale no smaller than this. The system is running with a 25% unallocated disk, to allow some free space for the learner to place moved blocks. The experiments use a set of traces and heuristic sets, described in the remainder of this section.

5.1 Traces

The evaluations in this paper use several traces. This describes each trace set.

92 Traces: These traces come from the Cello92 trace set [22]. This is a trace of the departmental server at HP in 1992, which served news, development, etc. The experiments use trace data from the users partition (disk B) and the news partition (disk D) from this server. We will refer to these traces as the *92 Users* and *92 News* traces.

99 Traces: These traces come from the Cello99 trace, a trace of the same system at HP in 1999. The experiments use trace data from the source partition (disk 1F002000), a partition containing development code (disk 1F015000), and a partition containing reference data (disk 1F003000.) We will refer to these traces as the *99 Source* , *99 Development* , and *99 Reference* traces.

5.2 Heuristics

The experiments found in this paper use the following combinations of heuristics.

All: This includes all of the heuristics listed in section 4.1.2.

All-no-bad: This includes all of the heuristics listed in section 4.1.2 except for the bad heuristic.

Placement: This includes the three placement heuristics: disk shuffling, front loading, and bad.

Sequential: This includes the two sequential heuristics: threading and run packing.

Thread+Shuffle: This includes the threading heuristic and the disk shuffling heuristic.

Thread+Placement: This includes the threading heuristic and two of the placement heuristics: disk shuffling and front loading.

6 Evaluating Overall Effectiveness

This section presents a representative evaluation of the learner, and discusses the weaknesses in our current methods. We will discuss the complexities of the problem, and our methods in more detail in later sections.

We run the learner on a full week from each trace. The learner trains on a given day, and then tests the performance on the next day. Because we are using a test window of 3 days, the first two days are purely

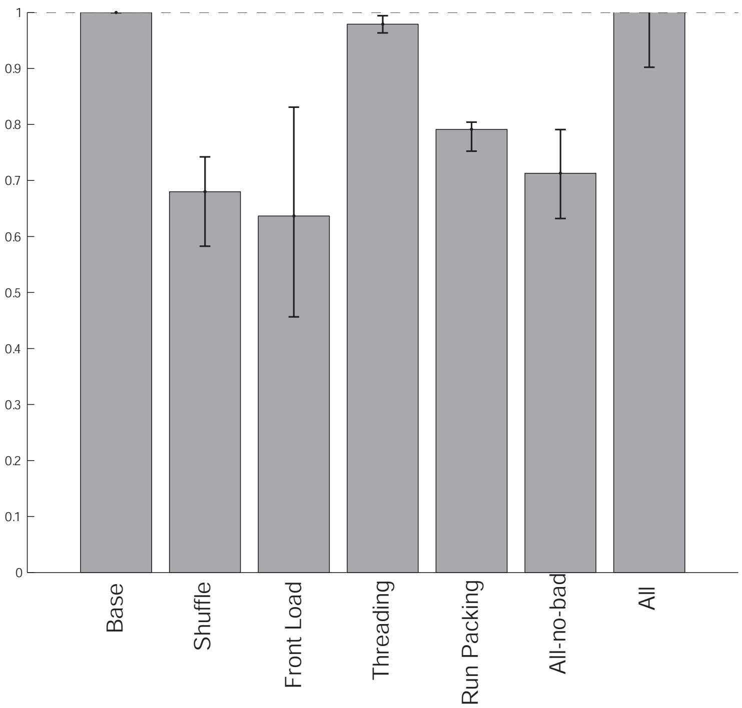


Figure 4: **Learner evaluation.** This evaluation shows results on the *92 News* trace, using the Apply learner described in more detail in Section 8.3.

training data. The results given are the average (with standard deviation bars) of the following five days. For each day the learning unit is run for 100 iterations.

We compare the results of the learner with all but the bad heuristic (All-no-bad) with the learner with all heuristics (All), and the base heuristics (Shuffle, Front load, Threading and Run Clustering.) The base case (Base) shows the performance of the layout when no reorganization is performed.

Figure 4 shows the results of these evaluations. The values shown in the graphs are average response times normalized to the base response time, so lower bars mean better performance. Note that these results do not include the cost of doing the actual reorganization.

A correctly performing learner should be as good or better than the best heuristic in all cases. The learner should also be able to ignore the bad heuristic; meaning that the addition of the bad heuristic will not hurt performance.. However, the figure shows that the learner without the bad heuristic (All-no-bad) does not do as well as Front Loading, which is the best heuristic. In addition, the learner is unable to completely filter out the bad heuristic; the learner with the bad heuristic (All) is significantly worse than the learner without the bad heuristic (All-no-bad).

We have tried a variety of approaches, and experimented with a variety of settings while tackling this problem. We have also evaluated the system on a variety of traces. While this exploration has yielded results which are sometimes better and sometimes worse, the graph shown is typical. The next section describes some of the reasons the learner does not perform as hoped.

7 Challenges

There are a variety of challenges in the adaptive combiner's task of finding a good layout given the constraints from the heuristics. This section discusses the most important of these challenges in detail.

7.1 Constraint Space

7.1.1 Constraint Conflicts

As mentioned earlier, conflicts are an important challenge that the learner must address. The complexity of the conflict space affects how well the greedy approach will work in applying constraints.

In order to explore this complexity, we performed an experiment measuring these conflicts. The experiment chooses 200 random constraints to monitor. For each of these constraints, it finds all combinations of a certain number of other constraints which, if applied, would keep the monitored constraint from being applied. Each of these combinations is listed as a single conflict. We will call the number of other constraints involved in the conflict the *conflict length*. Figure 3 on page 9 shows examples of conflicts. For example, the conflict between constraints A and B in the figure would be a conflict of length 1, because it includes one constraint besides A. The conflict between constraints A, C and D would be of length 2, since it includes two constraints besides A. This experiment looks for all conflicts of length 3 or smaller. We did not extend the search further because the methods of exhaustively searching for these conflicts are exponential and quickly become unmanageable (attempting to find conflicts of length 4 took more than 48 hours for a single day on some traces.)

Table 2 shows the average numbers of conflicts per constraint. It is interesting to note that as expected, larger numbers of heuristics usually lead to larger numbers of conflicts. Having both sequential and placement constraints also leads to higher numbers of conflicts, since the overlap allows for multiple placement constraints to conflict through a single sequential constraint. The *Thread+Shuffle* combination is interesting to note, as the two heuristics are complementary and have almost no conflicts.

Figure 5 shows the average number of conflicts per constraint across all of the trace sets by *conflict length*. There are a large number of conflicts of length one and two, but very few of length three. This is probably because with our heuristics, most conflicts occur between at most two placement constraints and

	92 Users	92 News	99 Source	99 Reference	99 Development
All	47.2	9.8	23.8	6.5	8.2
All-no-bad	26.7	5.2	9.1	3.4	5.0
Thread+Placement	7.1	1.5	1.5	3.0	1.1
Placement	1.9	1.9	2.0	1.9	1.9
Thread+Shuffle	0	0	0	0	0
Sequential	0.3	0.2	0.1	0.1	0.1

Table 2: **Conflict counts** This shows the average number of conflicts per constraint involving 3 or fewer other constraints.

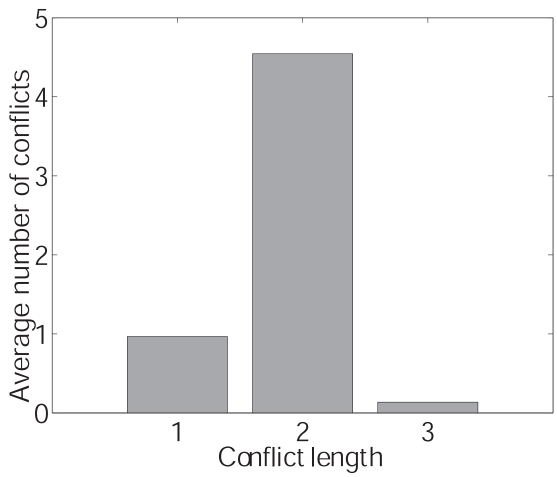


Figure 5: **Conflict order.** This graph shows the number of conflicts by conflict length. The number of conflicts is the average number for each constraint, averaged across all of the test cases found in Table 2.

a sequential constraint. Data from a subset of traces shows that the number of conflicts of length four is also low, and lower than the number of conflicts of length three. This suggests that most conflicts could be captured by a search depth of two or three.

7.1.2 Constraint Performance Dependencies

The performance of a constraint is also tied to its environment, creating another form of dependencies. Some number of other constraints will have an impact on the performance of each constraint. There are two main ways that a constraint can affect the performance of another constraint: overlap and access dependencies. Both depend upon the trace and the heuristics used.

7.1.3 Overlap Dependencies

Constraints may have dependencies caused by constraint overlap. In many cases, the learner can apply several constraints on the same set of blocks. When several constraints overlap in this way, the worth of a constraint can be affected by whether the other overlapping constraints are applied. For example,

Figure 6: **Overlap dependency.** This figure shows three overlapping constraints. Constraints A and B are sequential constraints, while constraint C is a placement constraint putting block 1 in disk area A. Any combination of these three constraints could be applied. However, the weight of each individual constraint will depend on what combination of other constraints are applied.

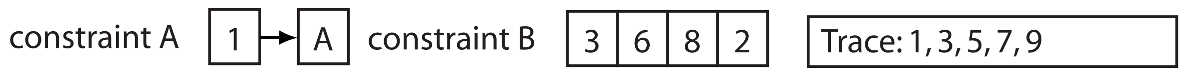


Figure 7: **Access dependency.** This figure shows an access dependency. Sequential constraint B is dependent on placement constraint A because constraint A affects a block which precedes a block in constraint B.

a placement constraint could heavily affect the performance of a sequential constraint affecting the same block by placing it in a different region. Figure 6 shows an example of an overlap dependency. We select those constraints which overlap with a constraint as dependent constraints. However, a constraint can also depend upon constraints which do not immediately overlap with it, but overlap with constraints that in turn overlap with the original constraint. These dependency chains can be arbitrarily long.

7.1.4 Access Dependencies

Constraints may also have dependencies which arise from access patterns in the trace. For example, if block a often follows block b in the trace, then the value of a constraint placing block a may depend upon the constraints placing block b. This dependency occurs because of the distance-dependent mechanical latencies of disk drives. Figure 7 shows an example of an access dependency.

Our model only considers the immediate predecessor in the trace as a possible dependency for simplicity. In the real system, however, any number of previous requests can affect the response time due to scheduling and caching algorithms. Ways to extend this notion of a predecessor could be an interesting area of study.

7.1.5 Evaluation

This section shows the results from a variety of experiments, all of which share a similar experimental set up. The experiments use the set of traces and heuristics described in Section 5. After generating the constraints from the set of heuristics given the trace, the experiments choose a random 1% of the constraints to monitor. They then monitor the change in constraint weight for these measured constraints under a given condition, over 30 runs. For each combination, the experiment runs on five days of trace. The reported values are the averages of these five days. The actual state of the experiment varies between the experiments and will be explained at that point.

The change in constraint weight is an interesting value because it shows how accurately we can predict

	92 Users	92 News	99 Source	99 Reference	99 Development
All	76%	67%	59%	48%	60%
All-no-bad	111%	87%	79%	59%	81%
Thread+Placement	96%	76%	76%	48%	72%
Placement	45%	41%	37%	30%	41%
Thread+Shuffle	119%	78%	87%	49%	84%
Sequential	35%	18%	36%	24%	34%

Table 3: **Performance dependencies** This shows the change in constraint weight when all of the dependencies are permuted.

	92 Users	92 News	99 Source	99 Reference	99 Development
All	5.4	3.8	4.4	4	4.2
All-no-bad	4.6	3	3.4	3	3.4
Thread+Placement	3	2	3	2	2.2
Placement	2	2	2	2	2
Thread+Shuffle	2	1	2	1	1.2
Sequential	2	1.6	2	1	1.2

Table 4: **Overlap counts** This shows the average number of dependent constraints using only overlap dependencies.

the weight of any individual constraint across runs. If this error is too high, it will be difficult for the learner to make decisions about constraints. To measure the change in constraint weights, we report percent standard deviation of the constraint weights.

The results from the first experiment are listed in Table 3. This table shows the standard deviation of the selected constraint weights when the overall layout is kept the same except for the dependent constraints for the selected constraints, which are randomly applied or not applied between the runs. The table shows that the standard deviation in constraint weight is quite high, running as high as 119%. This shows that these dependent constraints have a large effect on constraint weights, and must be considered.

Thus, in order to adequately evaluate constraint weight, we must determine what the weight of the constraint is in each of these conditions. This is a problem that is exponential in the number of dependent constraints.

Tables 4 and 5 show the average numbers of performance dependencies for each constraint. It is interesting to notice that the number of dependent constraints is fairly small on average, being around 10-15 constraints total. In general adding more heuristics increases the number of dependencies, but both the trace and the heuristics are important contributing factors in the number.

	92 Users	92 News	99 Source	99 Reference	99 Development
All	9.4	18.4	9	12	12.6
All-no-bad	8.2	15.6	7	9.2	10.2
Thread+Placement	6.2	11.4	6.2	7.2	7.8
Placement	5.6	12.6	5.8	8.8	8.6
Thread+Shuffle	4	6.8	3.8	4	4.8
Sequential	7.4	20.8	6	7.2	9.6

Table 5: **Dependency counts** This shows the average number of dependent constraints using both types of dependencies.

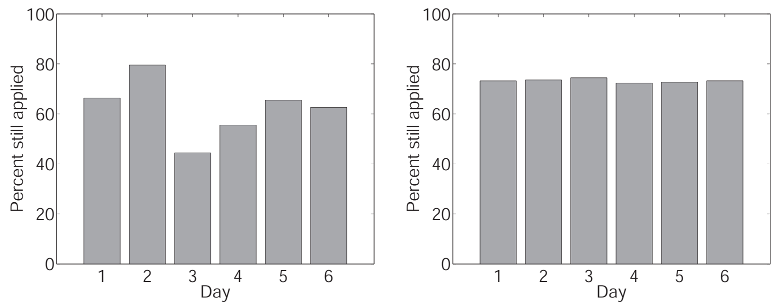
Heuristics	92 Users	92 News	99 Source	99 Reference	99 Development
All	23%	31%	18%	19%	21%
All-no-bad	30%	34%	15%	22%	23%
Thread+Placement	39%	40%	33%	24%	25%
Placement	35%	27%	31%	21%	22%
Thread+Shuffle	39%	46%	29%	29%	28%
Sequential	-	3%	11%	5%	6%

Table 6: **Conversion error** This shows the change in constraint weights even when the dependent constraints are pinned. The values not included did not generate enough constraints to perform the experiment.

7.1.6 Conversion Error

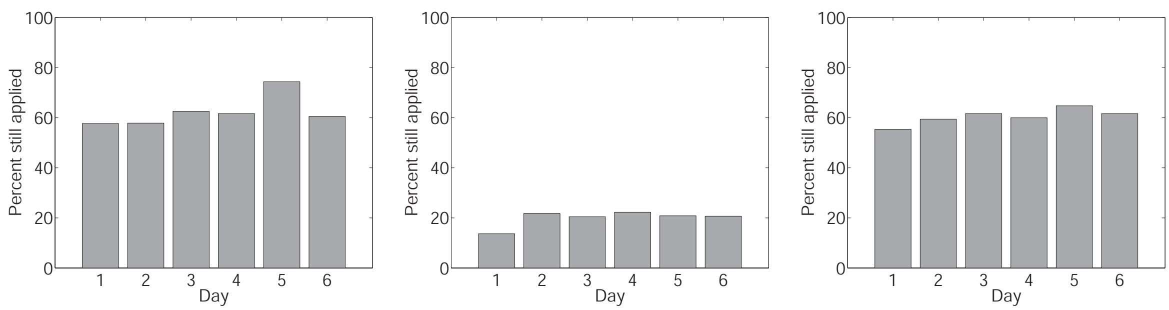
The conversion from constraint into layout also contributes some amount of randomness into constraint weights. Even when the set of dependent constraints discussed in the previous section are held constant, the performance of a constraint will be affected by other factors. For example, the placement constraints place a block in a given area, not an exact location. This means that the location of the block may change even if the constraints affecting that block do not. Other changes may affect cache behavior and scheduling decisions for a block as well. In addition, there may be other dependent constraints besides those we have chosen. For example, access dependencies can include more than just the immediate predecessor in the trace due to scheduling and caching. Overlap dependencies may include constraints which do not immediately overlap with a constraint, but overlap with constraints which then overlap with the original constraint. These dependency chains can be arbitrarily long.

Table 6 shows the conversion error for a few trace and heuristic sets. The conversion error is determined by applying all of the constraints the experiment has chosen to monitor. For each constraint upon which a monitored constraint depends, the experiment decides whether it will be applied at the first iteration, and keeps this constant throughout the runs. However, the other constraints in the set are randomly applied or not applied each run. This effectively shows the change in each constraint's weight if the constraints upon which it depends are held constant, but other constraints are varied.



(a) 92 Users.

(b) 92 News.



(c) 99 Source.

(d) 99 Reference.

(e) 99 Development.

Figure 8: **Constraint filtering** These graphs show the results from a long term experiment. In each experiment, a set of constraints is chosen by the learner to be applied on a given day. The learner then uses this set of constraints on the next 6 days. The graph shows what percent of these constraints the learner chooses to apply on the following 6 days. The graphs show that the change from day to day is usually not significant, but the change between workloads is large.

This standard deviation is higher than desired, in the 30% range - but should still be tolerable. Section 8 describes several tradeoff points that might help mitigate this conversion error. Expanding the dependency list might also decrease this error. However, the conversion error is much smaller than the dependent constraint error found in Table 3. Note that Table 3 does not include the conversion error, since the experiment only varies the dependent constraints, and not other constraints in the set.

7.2 Workload change

Another challenge in building this kind of system is that workloads change over time. This means that we must find constraints that will perform well over a long period of time, and not just on the current set of training data. Some constraints may only fit idiosyncrasies of the current day of trace, and may even be detrimental later on.

Figure 8 shows the results of a long term experiment. In this experiment, we first find a set of constraints which perform well on a given day. We then use this set of constraints for the next few days. The results show what percent of these original constraints the learner chooses to apply on the next 6 days. Each day was run with 10 iterations.

The graphs show that the percent applied is fairly steady from day to day, suggesting that most of the problem constraints are over-fitted to the particular day, rather than gradually made inappropriate by workload change. It is also interesting to note that the percent that are not applicable by the second day is considerable, between 20 and 80 percent. The large difference between different workloads is also noteworthy.

7.3 Problem Summary

The combination of the various problems listed above make creation of an effective learner very difficult. Unfortunately, solutions to one problem tend to make it difficult to solve the others. For example, the problem of solving which constraints to apply given a weight for each constraint is exponentially hard. However, the weight of each constraint is also dependent on other constraints, making the problem of finding the constraint weight exponential.

After finding a suitable solution to this challenge, the learner must take long term trends into account. However, when determining the value of constraints in a long term environment, the learner must remember the worth of the constraint is dependent upon which other constraints will also be applied, and which will not. This context may change from day to day, making it difficult to predict. These problem complexities, several of which were unexpected at the project's outset, explain why we were ultimately unable to construct a solid learner for our two-tiered architecture.

8 Design Tradeoffs

There are a wide variety of tradeoffs in the construction of this system. This section discusses a few of the tradeoffs that we have explored, for the benefit of researchers who pick up this line of research in the future.

8.1 Heuristic Support

Another area for exploration is the frequency of accesses needed for a constraint to be adequately evaluated. We will call the number of times the block is accessed support. If the blocks referenced by a constraint occur infrequently in the trace, the learner may not be able to determine the value of the constraint effectively. Initially we expected that low numbers of accesses would simple lead to weaker constraints. However, low numbers of accesses also mean that the constraint weight is more likely to be heavily influenced by chance factors. For example, if a block were only accessed a single time, whether the placement happened to fit that chance rotational latency would have a huge effect on the constraint weight. However, if the required

0	5	10	15	20
34%	36%	29%	29%	28%

Table 7: **Support experiment** This table shows the average variance of constraints when all dependent constraints were held pinned, but the remainder of the constraints varied. The jump from 5 to 10 is the biggest decrease in variance, prompting us to use the value of 10. The results show are from the use of all the heuristics but bad on the *92 News* trace.

support is too high, the heuristic will miss chances to optimize blocks which do not occur frequently enough to pass the threshold.

Table 7 shows experiments with different values of support on the 92 News trace. This is the same conversion variance experiment found in table 6, where the dependent constraints are held constant, but the remainder of the constraints are varied. However, this experiment is run on one trace and with different support thresholds. The experiment shows that a threshold of 10 decreased the amount of variance by a large amount, leading us to use a threshold of 10 accesses as the minimum to create a constraint. However, more detailed exploration of this area could be useful.

8.2 Performance Analyzer

There are a variety of tradeoffs in the performance analyzer. The current system uses a very high fidelity performance analyzer, namely trace-based simulation. However, it could use a variety of cheaper, simpler performance analysis methods as well.

It could be interesting to experiment with the effects of using the performance analyzer with or without caching, and with or without overlapping requests. While removing these factors decreases the accuracy of the model, it also decreases the complexity, making constraint weights more constant. If this does not significantly decrease the fidelity of the model, this may be advantageous.

8.2.1 Disk Areas

As described earlier, *place-in-area* constraints do not place blocks in specific locations, but in areas of the disk. The size of these disk areas has an effect on how the constraints are converted into an actual disk layout. A larger disk area means that it is possible to place large sequential constraints, and gives the learner more flexibility in applying changes.

However, large disk areas also increase the amount of variation possible while still satisfying constraints. If the areas are too large, the performance may depend heavily on the location in the area where the block is placed, which is not specified by the constraints. This would make the optimization method ineffective. Experimentation into "good" disk area sizes would be an interesting line of research. We chose to split the disk into 11 areas, along zone boundaries, as this seems like a size large enough to allow reorganization, but still large enough for choices in placement.

8.3 Learners

There are many ways to compute constraint potential, and we have experimented with several of them. Constraint potential attempts to quantify the usefulness of a constraint, and is calculated using constraint weights. The ideal value for the potential of a constraint is $P_c = R_{notc} - R_c$, where R_{notc} is the sum of the response times for the trace using the best layout that does not include c, and R_c is the sum of the response times for the trace using the best layout that does include c. Effectively, it is the advantage of including the constraint. Of course, this is not feasible to directly calculate, so each type of learner uses a different approximation.

8.3.1 Compare Learner

The *compare method* computes the constraint potential by comparing the constraint's weight against the other options available for a set of blocks. It first computes a per-block weight for the constraint by dividing the constraint weight by the number of blocks in the constraint. For each block in the constraint, it computes the per-block weight for all other constraints including the block. The constraint potential is then $P_c = \sum_{b \in B}(b_c - b_{other})$, where B is the blocks in the constraint, b_c is the per block weight, and b_{other} is the best weight for this block by another constraint.

While this method provides a relative value between options, it does not account for the possibility of several constraints being applied at the same time on the same set of blocks. In this case, the value of all of the constraints from the best set will converge to zero. While providing the appropriate contrast between overlapping blocks, it will not be useful to compare constraints on different sets of blocks, or different combinations of constraints. This prompted us to explore other combination options.

8.3.2 Application Learner

The *application method* takes the overlap of constraints into account. It keeps two weights for each constraint; an applied weight and a non-applied weight. The applied weight is set to the constraint weight when the constraint is applied. The non-applied weight is the sum of the weights of the blocks the constraint affects, when the constraint is not applied. The constraint potential is $P_c = W_u - W_a$, where W_u is the non-applied weight and W_a is the applied weight.

In effect, the potential is the difference between applying the constraint and not applying it. Constraints are applied if this value is positive, meaning that the application of the constraint decreases overall response time.

However, this approach does not adequately address the performance dependencies among the constraints. The weight for application and non-application is only a single weight, and does not account for the application state of dependent constraints. To address this aspect of the problem, we explored yet another learner.

8.3.3 Dependency Learner

The *dependency method* takes the general constraint performance dependencies into account. It keeps a list of all the dependent constraints for each constraint, as discussed in Section 7.1.2. Instead of keeping a single applied and non-applied weight, the learner keeps track of which dependent constraints were applied and whether the actual constraint was applied at each iteration.

The learner then sets the constraint potential to be the difference between the best value for the constraint when it is applied subtracted from the best value when it is not applied. The learner then sorts the list based on the potential, goes through the list, and only attempts to apply constraints that it predicts will perform well given the constraints that have already been applied.

A more thorough approach would reorder the constraints after each constraint is applied. It then only applies constraints that it predicts will do well in this situation. This would make the sorting step of the learner much more computationally intensive, but exploration of this option would be interesting.

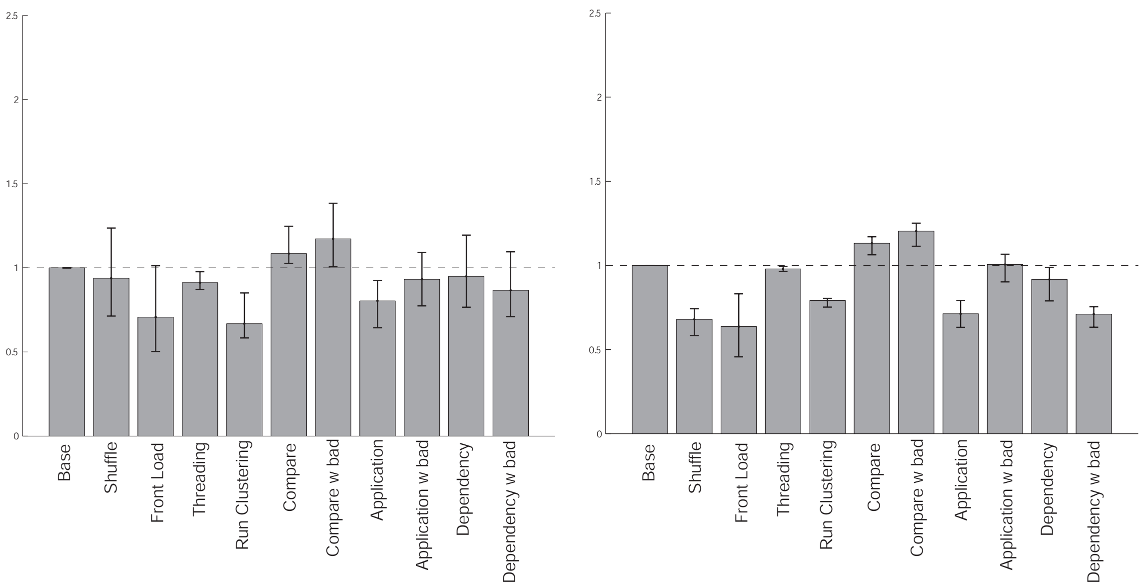
8.3.4 Evaluation

We evaluated each of the learning methods in comparison to individual heuristics. The experimental set up is the same as used in Section 6. However, we now look at all three of the learners.

We compare the results of each learner with all but the bad heuristic (Compare, Apply, Dependency), against each learner with all of the heuristics, including the bad heuristic (Compare w/ bad, Apply w/ bad, Dependency w/ bad) and the base heuristics (Shuffle, Front load, Threading and Run Clustering). All of the results are normalized to the base case, where no reorganization is performed (Base).

Figures 9 and 10 show the results of these evaluations. The values shown in the graphs are average response times normalized to the base response time, so lower bars mean better performance. Note that these results do not include the cost of doing the actual reorganization.

As expected, the application and dependency learners almost always perform better than the compare learner. However, sometimes the application learner performs better, and sometimes the dependency learner



(a) 92 Users.

(b) 92 News.

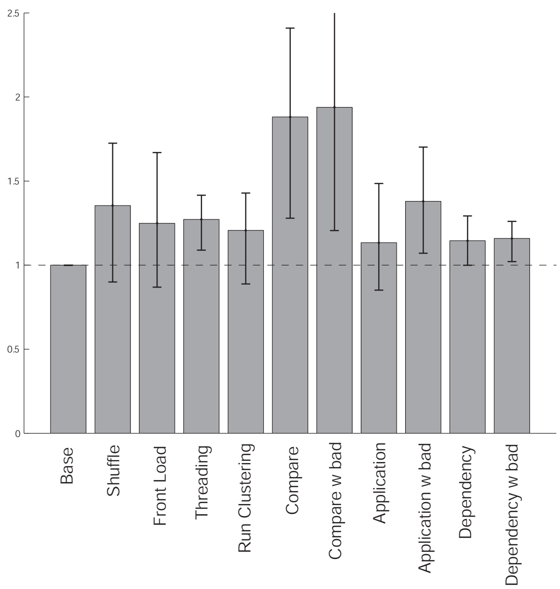
Figure 9: **Learner evaluation 92 traces** The evaluation results from the 92 traces. Note that none of the learners matches the best heuristic.

performs better. Because they both use non-optimal approaches, it is reasonable that one works better in some cases and another in others.

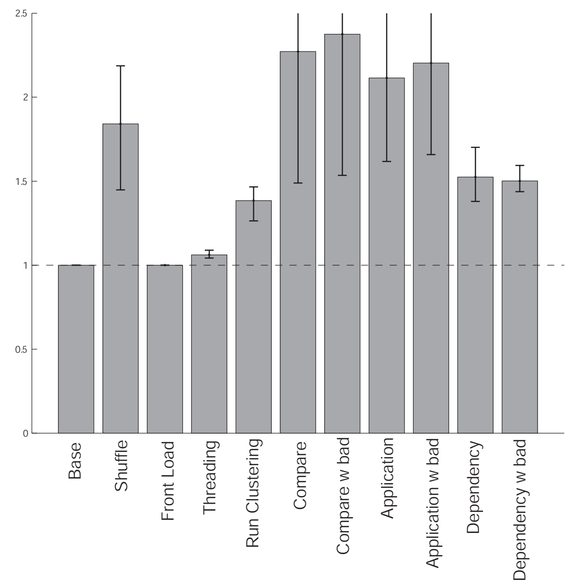
However, the dependency learner does handle the addition of the bad heuristic better than the other learners; this makes sense because the dependency learner is more aware of context. The bad heuristic is bad in two ways: first, it adds constraints which will perform poorly. Second, it makes other constraints look bad, causing the learner to discard what could be useful constraints. Because the dependency learner is careful about context, it can avoid discarding useful constraints due to influence from constraints from the bad heuristic.

It is also interesting to notice that sometimes the bad heuristic actually improves performance for the dependency learner. This may be due to the fact that the learner is better at determining interdependencies, and can thus pick out the occasional good suggestion among the bad. It is also possible that in some cases the spread of data that the bad heuristic performs may be beneficial in concert with the other heuristics.

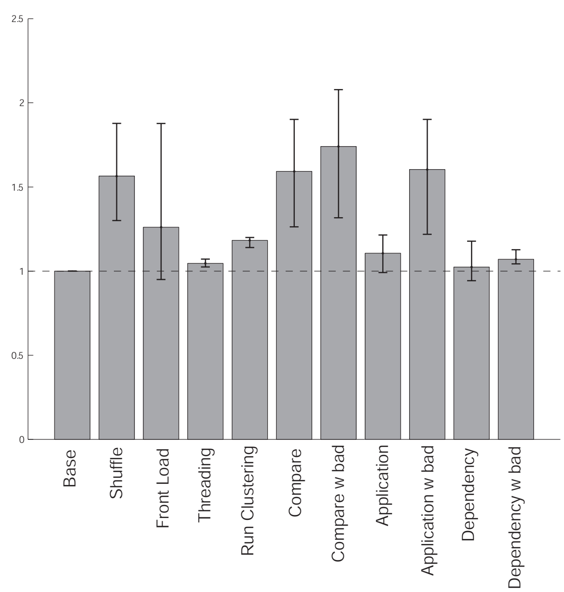
It is also interesting to note the difference between the Cello92 and Cello99 traces. The heuristics perform fairly well on the Cello92 traces, giving reasonable gains. However, the heuristics perform poorly on the Cello99 trace, usually causing degraded performance. This is probably due to the fact that the Cello99 traces are much larger, more intricate, and have a larger rate of change. The *99 Reference* trace is interesting in particular, because it is evident that the high rate of change in constraint worth showed in Figure 8 leads



(a) 99 Source.



(b) 99 Reference.



(c) 99 Development.

Figure 10: **Learner evaluation 99 traces** The results of the learner evaluations for the 99 traces. Note that none of the learners matches the best heuristic.

to poor performance of the learners.

Overall the learners do not perform very well. The learners are never better than the best heuristic, although they are sometimes close. In addition, the addition of the bad heuristic usually causes significant degradation in performance.

9 Future Work

There are a large number of future directions one could take with this work. We have tried several ad-hoc approaches to solving the learner problem, but a variety of other methods could applied, including genetic algorithms or other statistical methods. Extending the dependency learner to sort the list after placing each constraint would be one interesting option to explore. The fairly short lengths of conflicts suggest that an extension to the greedy method of conflict resolution which looks forward a few steps before placing each constraint may also be promising. In addition, thinking about the problem from a block-centric view point may lead to alternate optimization methods.

As mentioned in the paper, exploration into the granularity of the optimization would also be interesting. Doing reorganization at a file or other larger granularity may shrink the problem to a more manageable size.

The heuristic list could also be expanded; the current list is far from exhaustive. It would be interesting to see how the learner fares with more heuristics, including heuristics using knowledge of data structure, such as file system information. It could also be useful to further tune the existing heuristics, to improve their performance.

Additional exploration into the performance analyzer could also be useful. The current analyzer is very computationally intensive, limiting the number of iterations that can be performed. A faster, less accurate performance analyzer might yield better results due to its ability to look at more options. Some methods of performance analysis may also allow efficient evaluation of individual constraint addition or removal, allowing a more targeted approach to evaluating constraint value. More exploration into the optimal disk size could also be useful.

10 Conclusion

The concept of a two-tiered learning architecture provides the promise of a flexible and robust way of combining optimization approaches. However, building a learning agent for such a system has proved difficult. Unfortunately, constraints are not independent; significant dependency exists both in the value of constraints and in conflicts between constraints. Our approaches have had some nominal success, but better approaches will be necessary for a full solution. We have outlined a structure for a two-tiered learning architecture for disk layout, analyzed the key challenges in building such a system, and suggested directions for further research. This research also suggests a higher level point; that the problem of policy automation is difficult in practice.

11 Acknowledgements

I would like to thank my advisor Greg Ganger. I would also like to thank the many people who have helped me with ideas, discussions, and writing code. I especially thank Craig Soules, Eno Thereska, John Strunk, James Hendricks and Shobha Venkataraman. I am funded by an NSF fellowship.

References

[1] Sedat Akyurek and Kenneth Salem. Adaptive block rearrangement. *ACM Transactions on Computer Systems*, **13**(2):89–121. ACM Press, May 1995.

[2] Ahmed Amer, Darell Long, Jehan-Francios Paris, and Randal Burns. File access prediction with adjustable accuracy. *International Performance Conference on Computers and Communication* (Phoenix, AZ, April 2002). IEEE, 2002.

[3] Ismail Ari, Ahmed Amer, Robert Gramacy, Ethan L. Miller, Scott A. Brandt, and Darrell D. E. Long. ACME: Adaptive Caching Using Multiple Experts. *Workshop on Distributed Data and Structures* (Paris, France, March 2002), 2002.

[4] John S. Bucy and Gregory R. Ganger. *The DiskSim simulation environment version 3.0 reference manual*. Technical Report CMU–CS–03–102. Department of Computer Science Carnegie-Mellon University, Pittsburgh, PA, January 2003.

[5] Gregory R. Ganger and M. Frans Kaashoek. Embedded inodes and explicit grouping: exploiting disk bandwidth for small files. *USENIX Annual Technical Conference* (Anaheim, CA), pages 1–17, January 1997.

[6] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley.

[7] James Griffioen and Randy Appleton. Reducing file system latency using a predictive approach. *Summer USENIX Technical Conference* (Boston, MA, June 1994), pages 197–207. USENIX Association, 1994.

[8] James Griffioen and Randy Appleton. *The design, implementation, and evaluation of a predictive caching file system*. Technical Report CS–264-96. University of Kentucky, June 1996.

[9] Windsor W. Hsu, Alan J. Smith, and Honesty C. Young. The automatic improvement of locality in storage systems. Computer Science Division, University of California, Berkeley, TR CSD–03–1264, July 2003.

[10] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, and Deborah A. Wallach. Server operating systems. *ACM SIGOPS. European workshop: Systems support for worldwide applications* (Connemara, Ireland, September 1996), pages 141–148. ACM, 1996.

[11] Thomas M. Kroeger and Darrell D. E. Long. Predicting file system actions from prior events. *USENIX Annual Technical Conference* (San Diego, CA, 22–26 January 1996), pages 319–328. USENIX Association, 1996.

[12] Nick Littlestone and Manfred K. Warmuth. *The weighted majority algorithm*. UCSC–CRL–89–16. DEPTCS,. University of California at Santa Cruz, July 1989.

[13] Sai-Lai Lo. *Ivy: a study on replicating data for performance improvment*. TR HPL-CSP-90-48. Hewlett Packard, December 1990.

[14] Tara M. Madhyastha and Daniel A. Reed. Input/output access pattern classification using hidden Markov models. *Workshop on Input/Output in Parallel and Distributed Systems* (San Jose, CA), pages 57–67. ACM Press, December 1997.

[15] Tara M. Madhyastha and Daniel A. Reed. Intelligent, adaptive file system policy selection. Frontiers of Massively Parallel Computation, October 1996.

[16] Scott McFarling. *Combining branch predictors*. Technical Report TN-36. Digital Western Research Lab., 1993.

[17] Tom M. Mitchell. *Machine learning*. McGraw-Hill, 1997.

[18] Spencer W. Ng. Improving disk performance via latency reduction. *IEEE Transactions on Computers*, **40**(1):22–30. IEEE, January 1991.

[19] James Oly and Daniel A. Reed. Markov model prediction of I/O requests for scientific applications. *ACM International Conference on Supercomputing* (New York, NY, June 2002). ACM, 2002.

[20] L. R. Rabiner. A tutorial on hidden Markov models and selected applications in speech recognition. Published as *Proceedings of the IEEE*, **77**:227–248, 1989.

[21] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, **10**(1):26–52. ACM Press, February 1992.

[22] Chris Ruemmler and John Wilkes. UNIX disk access patterns. *Winter USENIX Technical Conference* (San Diego, CA, 25–29 January 1993), pages 405–420, 1993.

[23] Chris Ruemmler and John Wilkes. *Disk Shuffling*. Technical report HPL-91-156. Hewlett-Packard Company, Palo Alto, CA, October 1991.

[24] Brandon Salmon, Eno Thereska, Craig A. N. Soules, and Gregory R. Ganger. A two-tiered software architecture for automated tuning of disk layouts. *Algorithms and Architectures for Self-Managing Systems* (San Diego, CA, 11 June 2003), pages 13–18. ACM, 2003.

[25] Jiri Schindler and Gregory R. Ganger. *Automated disk drive characterization*. Technical report CMU–CS–99–176. Carnegie-Mellon University, Pittsburgh, PA, December 1999.

[26] Jiri Schindler, John Linwood Griffin, Christopher R. Lumb, and Gregory R. Ganger. Track-aligned extents: matching access patterns to disk drive characteristics. *Conference on File and Storage Technologies* (Monterey, CA, 28–30 January 2002), pages 259–274. USENIX Association, 2002.

[27] Elizabeth Shriver, Eran Gabber, Lan Huang, and Christopher A. Stein. Storage management for web proxies. *USENIX Annual Technical Conference* (Boston, MA, 25–30 June 2001), pages 203–216, 2001.

[28] T-YYeh and Y. N. Patt. Two-level adaptive branch prediction. *24th ACM/IEEE International Symposium and Workshop on Microarchitecture* (November 1991), pages 51–61, 1991.

[29] Jun Wang and Yiming Hu. PROFS – Performance-Oriented Data Reorganization for Log-structured File System on Multi-Zone Disks. *Ninth International Symposium on Modeling, Analysis and Simulation on Computer and Telecommunication Systems* (Cincinnati, OH, August 2001), pages 285–293, 2001.

[30] Xiang Yu, Benjamin Gum, Yuqun Chen, Randolph Y. Wang, Kai Li, Arvind Krishnamurthy, and Thomas E. Anderson. Trading capacity for performance in a disk array. *Symposium on Operating Systems Design and Implementation* (San Diego, CA, 23–25 October 2000), pages 243–258. USENIX Association, 2000.